Cincom Smalltalk™ ObjectStudio®
User Interface Guide

Publication Number P40-3811-03

All Rights Reserved

CINCOM SYSTEMS, INC. MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE MATERIAL
CONTAINED IN THIS MANUAL, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The information contained in this
manual is subject to change without notice. This manual contains proprietary information that
is protected by copyright. All rights are reserved. It may not be photocopied, reproduced, or
translated, in whole or in part, without the prior express written consent of Cincom Systems,
Inc.

See http://www.cincom.com/legal/terms.html for a list of Cincom trademarks and other
trademarks that may appear in Cincom product documentation.

All other trademarks are trademarks or registered trademarks of their respective companies.

Cincom Systems, Inc.
55 Merchant Street
Cincinnati, Ohio 45246-3732
USA

PHONE: (513) 612-2300
FAX: (513) 612-2000
WORLD WIDE WEB: http://www.cincom.com

Attention:

Some Cincom products, programs, or services referred to in this publication may not be
available in all countries in which Cincom does business. Additionally, some Cincom products,
programs, or services may not be available for all operating systems or all product releases.
Contact your Cincom representative to be certain the items are available to you.
Release information for this manual


Cincom Technical Support for ObjectStudio

<table>
<thead>
<tr>
<th></th>
<th>Web:</th>
</tr>
</thead>
<tbody>
<tr>
<td>All customers</td>
<td><a href="http://supportweb.cincom.com">http://supportweb.cincom.com</a></td>
</tr>
<tr>
<td>USA customers</td>
<td>Phone:</td>
</tr>
<tr>
<td></td>
<td>1-800-727-3525</td>
</tr>
<tr>
<td></td>
<td>Fax:</td>
</tr>
<tr>
<td></td>
<td>(513) 612-2000</td>
</tr>
<tr>
<td></td>
<td>Attn: ObjectStudio Support</td>
</tr>
<tr>
<td></td>
<td>Mail:</td>
</tr>
<tr>
<td></td>
<td>Cincom Systems, Inc.</td>
</tr>
<tr>
<td></td>
<td>Attn: ObjectStudio Support</td>
</tr>
<tr>
<td></td>
<td>55 Merchant Street</td>
</tr>
<tr>
<td></td>
<td>Cincinnati, OH 45246-3732</td>
</tr>
<tr>
<td></td>
<td>USA</td>
</tr>
<tr>
<td>Outside of USA</td>
<td>All:</td>
</tr>
<tr>
<td></td>
<td>Visit the support links at <a href="http://www.cincom.com">http://www.cincom.com</a> to find contact information for your nearest Customer Service Center.</td>
</tr>
</tbody>
</table>

Document description

This guide is for beginner and advanced users of ObjectStudio® who want to use the Designer to create interfaces. For more advanced information about programming tools and designing object-oriented applications, refer to the *ObjectStudio User's Guide*, P40-3807.
# Contents

1. Introducing the Designer ............................................................... 11
   What is the Designer? ................................................................................... 11
   Automatic code generation ....................................................................... 12
   Typical uses ......................................................................................... 12
   Designer concepts ....................................................................................... 13
      Form .................................................................................................. 14
      Interface ............................................................................................. 15
      Controller ............................................................................................ 15
      Form item ........................................................................................... 16
      Controller item ..................................................................................... 16
      Interface component .............................................................................. 17
   Designer basics .......................................................................................... 18
      Opening the Designer .............................................................................. 18
      Designer layout ..................................................................................... 20
      Designer modes ..................................................................................... 23
      Exiting the Designer ................................................................................ 24
   Designer menu overview ............................................................................... 25
      File menu ............................................................................................ 25
      Controller menu .................................................................................... 26
      Component menu ................................................................................... 26
      Form menu .......................................................................................... 27
      FormItem menu ..................................................................................... 28
      Tools menu .......................................................................................... 29
      Options menu ....................................................................................... 30
      Help menu ........................................................................................... 30
   Quick tour ................................................................................................ 31
      Starting the Designer .............................................................................. 32
      Creating the Static Text item ............................................................... 34
      Creating the Entry Field item .............................................................. 35
      Creating the Greet button ....................................................................... 36
      Creating the Done button ....................................................................... 37
      Creating the greetMe method ............................................................... 39
      Linking the method to a Button ............................................................ 41
      Saving changes ...................................................................................... 42
      Testing the interface .............................................................................. 43
2. Creating forms ............................................................... 45

Working with forms ........................................................... 45
  Creating a form ............................................................... 45
  Setting focus to a form ..................................................... 47
  Clearing a form .............................................................. 47
  Deleting a form .............................................................. 48
  Changing a form ............................................................. 49
  Using the default tab order on a form or specifying your own .......... 55

Modal dialog boxes ............................................................ 59
  Types of modal dialog boxes ............................................. 59
  Synchronism ................................................................. 60
  Modal message boxes .................................................... 60
  Creating a modal dialog box ............................................. 60

Using the Method Editor to create a method .......................... 61
  Opening the Method Editor .............................................. 61
  Method Editor buttons .................................................... 62
  Method Editor pop-up menu ............................................. 62
  Creating and editing methods in the Method Editor ............... 63
  Method Editor shortcut list boxes and buttons ....................... 64
  Automation Function Helper .......................................... 65
  Creating a method reference in the Method Editor ............... 70

Naming an interface ............................................................ 71

Saving an interface ............................................................ 72

Testing the interface ............................................................. 74

Menu editor .......................................................................... 75
  Getting started ............................................................... 76
  Creating a menu bar ......................................................... 79
  Menu Item Options .......................................................... 80
  Menu Options .................................................................... 83
  Creating pull-down menu items ......................................... 84
  Menu Definition list box .................................................... 86
  Creating a cascading menu ............................................... 89
  Creating a pop-up menu .................................................... 90
  Adding context-sensitive help to an item ............................... 95
  Assigning actions to menu items ....................................... 95

Toolbar editor ......................................................................... 98
  Creating a toolbar ........................................................... 98
  Editing a toolbar ........................................................... 99
  Creating a toolbar example .............................................. 104
  Associating the toolbar with methods ................................. 106

Status line editor .................................................................... 107
  Creating a status line ....................................................... 107
  Editing a status line ........................................................ 109

Tooltip editor ......................................................................... 113
  What are Tooltips? .......................................................... 113
  Creating tooltips for form items ....................................... 113
Creating property pages and property sheets ..................................................... 114
Creating a property page ........................................................................ 115
Creating a property sheet ........................................................................ 116
Adding property pages to a property sheet ................................................... 117
Displaying the property sheet ................................................................... 118

3. Creating form items ................................................................ 120
Form item overview.................................................................................... 120
Working with form items ........................................................................ 125
Creating a form item ........................................................................ 125
Creating instance variables ................................................................ 128
Selecting a form item ........................................................................ 129
Deleting a form item ........................................................................ 129
Using the grid .................................................................................. 130
Resizing a form item ........................................................................ 131
Placing a form item .......................................................................... 132
Aligning form items ........................................................................ 133
Changing the class of a form item ......................................................... 136
Customizing form items ................................................................................ 137
Customizing multiple form items ............................................................... 138
Creating mnemonics for form items ........................................................... 139
Changing the format of a form item ......................................................... 139
Declaring the source of data for a form item ................................................. 146
Assigning methods to form items .............................................................. 150
Setting form item attributes .................................................................... 152
Moveable Borders .................................................................................. 156
Enabling or disabling smart list search and smart list sounds ......................... 160
Form item descriptions ................................................................................ 163
Bitmap Selector .................................................................................... 163
Button ............................................................................................... 165
Chart ................................................................................................ 167
Check Box ........................................................................................... 172
Check List ........................................................................................... 173
Combo List .......................................................................................... 174
Date Field ........................................................................................... 175
Draw Pad ............................................................................................ 175
Drop-down Combo Box ........................................................................... 185
Drop-down List Box ............................................................................... 186
Entry Field .......................................................................................... 187
Horizontal Border ................................................................................. 190
Line .................................................................................................. 190
List Box .............................................................................................. 190
ListView ............................................................................................. 193
Multi Line Entry Field ........................................................................... 208
Notebook ........................................................................................... 208
Number Field ..................................................................................... 211
OLE Custom Control ................................................................. 215
OLE Item .................................................................................. 215
Progress Bar .............................................................................. 216
Radio Buttons ............................................................................ 218
Rectangle .................................................................................. 219
Scatter Graph ........................................................................... 223
Slider ........................................................................................ 224
Spin Button .............................................................................. 225
Static Text ................................................................................ 226
Tabular List Box ....................................................................... 226
Text collector ........................................................................... 231
Text Editor ............................................................................... 233
Topic Box ................................................................................ 233
Tree View ............................................................................... 234
Value Set ................................................................................ 242
Vertical border ........................................................................ 244
Workplace List Handler ............................................................. 244
Workplace Object Handler ......................................................... 245
Workplace Object Template ....................................................... 245

4. Working with interfaces and interface components .............. 246

Interfaces and interface components .................................................. 246
Components of a UI item .................................................................. 246
Using a template to create an interface ................................................. 247
Making an interface into a template .................................................... 250
Using a template ........................................................................ 252
Creating an interface component ....................................................... 253
Creating a font selector ................................................................ 253
Making the component reusable ......................................................... 259
Saving and testing your work ............................................................. 261
Adding an interface component to an interface ................................... 262
Using the font selector in an interface ............................................... 263
Creating new interface form items ....................................................... 266
Creating the form item .................................................................. 267
Creating the controller item ............................................................. 268
Using the new user interface item ..................................................... 269
Working with instance variables ......................................................... 270
Adding an instance variable ............................................................. 271
Multiple document interface ............................................................ 272
Creating an MDI application ............................................................ 272
Working with MDI menus ............................................................... 273
Adding a Window menu to an MDI form ............................................. 275
5. Creating a workplace application ......................................................... 277

What is a workplace application? ................................................................. 277
Using a workplace application ................................................................. 278
Workplace form items ............................................................................ 279

Creating a workplace application ................................................................. 280
Creating the main form ............................................................................ 281
Adding form items .................................................................................... 281
Changing form items ............................................................................... 282
Creating the editor form ......................................................................... 283
Creating an instance variable .................................................................... 284
Subclassing the accepts:in: method ............................................................. 285
Creating the in:drop:at: method ................................................................. 286
Creating the doOk method ....................................................................... 286
Assigning the doOk method ...................................................................... 286
Assigning a return key method .................................................................. 287
Assigning a form to an object handler ......................................................... 288
Testing the interface ............................................................................. 289
Adding a pop-up menu ............................................................................ 290

Creating a Workplace Template Object ..................................................... 292
Creating a new class to define the template ................................................. 292
Creating a workplace template ................................................................. 293
Testing the workplace template ................................................................. 294

6. Using the Model Editor ............................................................................ 295

Introduction ............................................................................................. 295
Description of the Model Editor ................................................................. 295
Opening the Model Editor ............................................................................ 296
Defining variables ..................................................................................... 297
Creating variables ..................................................................................... 297
Assigning values to variables .................................................................... 298
Setting up a graph .................................................................................... 299
Linking models and graphs to interfaces ................................................... 300

7. Programming graphical user interfaces .................................................. 301

Visual interface objects ............................................................................ 301
Form and form item geometry .................................................................. 303
Controller message protocol ..................................................................... 306
Initializing an interface ............................................................................ 306
Opening a form ...................................................................................... 307
Setting focus to a form ............................................................................ 308
Closing a form ...................................................................................... 308
Method summary .................................................................................... 309
8. Responding to user interaction .................................................. 310

Working with events ................................................................. 310
  Form events ............................................................................. 310
  Form item events ..................................................................... 314
  General events ......................................................................... 317
Using the Model View Controller approach ............................................. 324
  How MVC works ....................................................................... 326
  Using the Designer .................................................................... 327
  Writing MVC code ...................................................................... 330
  Writing MVC code: Example overview ......................................... 344
  MVC example: Part 1 ................................................................. 345
  MVC example: Part 2 ................................................................. 351
  MVC example: Part 3 ................................................................. 354
Validating data .............................................................................. 357
  Overview of the process ............................................................ 357
  Default validation ...................................................................... 358
  Validating data in FormString items .......................................... 359
  Validating data at the controller level ......................................... 361
  Validating data at the controller item level .................................... 362

9. Extended interface example ..................................................... 363

Introduction .................................................................................. 363
Overview of the example .................................................................. 363
Building the user interface ............................................................... 365
  Building the main form ............................................................ 365
  Building the Employee dialog box ............................................. 369
  Initializing forms ...................................................................... 372
  Initializing the main form ........................................................ 374
  Testing your work ..................................................................... 378
Sorting the table ........................................................................... 378
  Using the order: method to sort a table ...................................... 379
  Linking the Sort radio buttons .................................................. 380
  Linking the sort menus ............................................................. 382
  Testing your work ..................................................................... 383
Synchronizing radio buttons and menus .............................................. 384
  Creating methods to synchronize Buttons and menus .................. 384
  Avoiding the endless loop ........................................................ 386
  Using flags to synchronize updates ............................................ 388
  Testing your work ..................................................................... 390
Implementing the Delete option .......................................................... 391
  Confirming the deletion ............................................................ 391
  Deleting the record ................................................................... 392
  Assigning a method to the Delete button ..................................... 392
  Testing your work .................................................................... 392
Displaying the Employee dialog box ................................................................. 393
  Opening the dialog box ........................................................................... 393
  Initializing the dialog box ........................................................................ 394
  Testing your work ................................................................................. 396
Accepting and saving changes ........................................................................ 397
  Updating a record ................................................................................. 397
  Inserting a record ................................................................................. 398
  Dynamically changing the Accept method .................................................... 400
  Testing your work ................................................................................. 400
Implementing the other Buttons ..................................................................... 401
  Close button ........................................................................................ 401
  Previous button .................................................................................... 402
  Next button ........................................................................................ 402
  Testing your work ................................................................................. 403
Optional enhancements .......................................................................... 403
Summary ................................................................................................. 404

Index ........................................................................................................ 405
1. Introducing the Designer

What is the Designer?

The Designer is a graphical tool that helps you create a graphical user interface (GUI) for your application. Specifically, it provides:

♦ An extensive set of predefined graphical form items, such as Buttons, text boxes, and Sliders, that you can drag and drop onto your interface

♦ Numerous tools that format items, define key assignments, declare the order of item traversal, and perform other functions

♦ A menu tool that allows you to create pull-down, cascading, and pop-up menus

♦ Tools for creating toolbars and status lines

♦ A Method Editor to use within the Designer

♦ A point-and-click facility for linking items on the interface to predefined methods or to methods that you define

♦ A point-and-click facility that links display items to a data source; for example, a query, a database, or a financial model

♦ An integrated test facility that allows you to test the interface without exiting the Designer
Automatic code generation

When you save your Designer work, ObjectStudio automatically generates the Smalltalk source code that implements the interface.

Typical uses

Typically, when you work on a user interface, you use a combination of graphical interaction with the Designer and programming Smalltalk code. The following table presents general guidelines about when to use each approach:

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
<th>For example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designer</td>
<td>Create windows, arrange items in the window, format windows and items, create links to data and behavior.</td>
<td>Create a window, arrange an Entry Field and two Buttons in it, assign each Button a color, and link one Button to a method.</td>
</tr>
<tr>
<td>Smalltalk code</td>
<td>Describe behavior that arises in response to user interaction.</td>
<td>Describe how the system reacts when a user clicks a Button or selects a menu item.</td>
</tr>
</tbody>
</table>

As you become more familiar with Smalltalk and the Designer, you will develop your own style of working with each.
Designer concepts

As you work with the Designer and as you write Smalltalk code to enhance your interface, you will need to understand the concepts listed here, and which are discussed in the sections that follow:

♦ Form
♦ Interface
♦ Controller
♦ Form item
♦ Controller item
♦ Interface component

This section assumes that you are familiar with the basics of Smalltalk and host windowing systems. You can read more about both topics in the *ObjectStudio Smalltalk User’s Guide*, P40-3810.
Form

A form is a GUI window. There is a special form, the main form, which by default is the first form you create. All other forms in the interface depend on it. When a user opens the interface, this form is displayed. Several types of forms are described in the following table:

<table>
<thead>
<tr>
<th>Form type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>By default, a form is moveable and sizeable. It has a title bar and a System Menu, and it can have a menu bar, toolbar, and status line.</td>
</tr>
<tr>
<td>Subform</td>
<td>A form that has the characteristics of an MDI (Multiple Document Interface) child window. Removing the title bar and System Menu from a subform removes the subform’s other MDI characteristics. A subform cannot have a menu bar.</td>
</tr>
<tr>
<td>Dialog box</td>
<td>A form that is moveable. It is not sizeable and cannot have a menu.</td>
</tr>
<tr>
<td>Property page</td>
<td>A form that can be added to a property sheet, which is a collection of property pages.</td>
</tr>
<tr>
<td>Modal dialog box</td>
<td>A dialog box that, when displayed, temporarily prevents the user from using the application. The user must first interact with and close the modal dialog box before continuing to use the application.</td>
</tr>
<tr>
<td>System modal dialog box</td>
<td>A dialog box that, when displayed, prevents the user from interacting with the operating system. On Windows 95 and NT® 4.0, you can do other tasks, but the dialog box remains on top until you close the dialog box.</td>
</tr>
</tbody>
</table>

Designer: You can use the Designer to create, arrange, and set characteristics of a form. You also can use the Designer to create a menu bar, toolbars, and status lines for a form.

Smalltalk: In Smalltalk, each form is represented by an instance of class Form. Specialized types of forms are represented by subclasses of Form, for example, DialogBox and SubForm. An instance of class Form tracks items that display on the form.
Interface
Each time you open an empty version of the Designer, you create a new interface. An interface can contain one or more forms, and, of course, each form can contain many items.

An interface is an abstract concept in ObjectStudio; it does not have a corresponding Smalltalk representation.

Controller
Each interface has a corresponding controller. As the user interacts with the interface, the system sends messages to the controller. In turn, the controller sends messages to components of the interface. In other words, the controller manages communications between:

♦ The user and the interface
♦ The interface and the business layer of the application

Designer and Smalltalk: When you save work in the Designer, ObjectStudio generates Smalltalk code, including the implementation of the controller. For each interface that you create, ObjectStudio creates a subclass of the Controller class. To implement your interface, ObjectStudio also creates an instance of the subclass it just created.

Each controller tracks information about the interface, including:

♦ Forms in the interface
♦ Controller items used by the interface (see “Controller item” on page 16)
♦ Types of user events that can affect the interface
Form item

Each item that you can place in a form is called a *form item*. Some form items are very simple; such as Buttons and Static Text form items. Others, such as Text Editors and Value Sets, are more complex.

**Designer:** “Quick tour” on page 31 describes how you can use the Designer to create form items and arrange them on a form. You can use the Designer to change form items in many ways. For example, you can use the Designer to:

♦ Change how a form item displays
♦ Link a form item to a data source
♦ Assign a method to execute in response to a user action

**Smalltalk:** In Smalltalk, each type of form item (for example, Button, Slider, or Value Set) is represented by a subclass of theFormItem class (for example, FormButton, FormSlider, FormValueSet). Each form item you create is an instance of the appropriate subclass ofFormItem.

Controller item

Whenever you add a new form item to a form, ObjectStudio creates an instance of a subclass ofFormItem and an instance of a subclass ofControllerItem. In general, a controller item manages the behavior of an item by:

♦ Managing initial values
♦ Setting and getting values
♦ Responding to user interaction

For example, a controller item is responsible for reacting to a mouse click on an item.

When the user interacts with an interface, the interface’s controller sends a message to the appropriate controller item, and in turn, the controller item sends a message to the corresponding form item.

**Smalltalk:** As with form items, ObjectStudio represents each type of item as a subclass ofControllerItem. For example, the ControllerItem subclasses for Button, Slider, and Value Set are ButtonCtrl, SliderCtrl, and ValueSetCtrl.
**Interface component**

An *interface component* is a collection of form items that you can create with the Designer and reuse in multiple forms or interfaces. For more information about interface components, see “4. Working with interfaces and interface components” on page 246.

**Example:** Imagine that your application asks for a person’s U.S. social security number and U.S. postal code on several different forms. Parts of the application might need this information to enter a new database record, to verify a personnel record change, and to do a quick database search.

**Designer:** Using the Designer, one approach to creating each form is to create a form item for each component. So for each form, you would create two Static Text boxes for labels and two Entry Fields for the U.S. social security number and U.S. postal code. Then, you would arrange the items on the form, format each item, and possibly describe the type of data expected in the Entry Fields.

Rather than redo this work for each form, you can create an interface component that creates the four form items described. Then, you can place the interface component on the three forms just as you would place a predefined form item.

Advantages of this approach are:

♦ Maintain consistency across an application when you present and request similar types of information

♦ Reuse work that you do in the Designer

**Smalltalk:** When you save work from the Designer, ObjectStudio creates a subclass of the class InterfaceComponent that corresponds to each interface component that you defined. For each interface component, it also creates an instance of the appropriate subclass; this instance implements the interface component.
Designer basics

This section shows you how to use the Designer.

Opening the Designer

You can open the Designer to create a new interface or to edit an existing interface. To open the Designer, start at the ObjectStudio Desktop.

Creating a new interface

To create a new interface:

1. Click the Create New Interface button (shown at left) on the Desktop toolbar. You can also select File > New > Interface menu option from the Desktop menu.

The Select Controller Type dialog box displays, as shown here:

2. Click OK on the Select Controller Type dialog box (accept the default controller item from the List Box) to open the Designer.
Editing an existing interface

To edit an existing interface:

1. Select **File > Load**, then specify the name of the file that you want to edit and click **OK**.
   
   ObjectStudio displays the icon of the loaded file on the Desktop.

2. On the Desktop, select the icon of the interface you want to edit; then select **Edit** from the pop-up menu or select the icon of the interface that you want to edit; then select **Edit > Edit**.
   
   The Designer opens.

For more information

To learn more about working with interface components or about the Select Controller Type dialog box, see “4. Working with interfaces and interface components” on page 246.
Designer layout

When you open the Designer, the Designer window displays with these components:

♦ Menu bar
♦ Toolbar
♦ Alignment toolbar, which helps you align form items
♦ Status line, which provides hints as you work in the Designer

A main form, which has separate focus from the Designer window, also displays. You can minimize, maximize, or close the main form by using the Buttons on the title bar. The following figure shows the Designer when you first open it using the default controller:
Menu bar

The menu bar contains the following Designer menus:

<table>
<thead>
<tr>
<th>Menu</th>
<th>Allows you to</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Save and test your work and exit the Designer.</td>
</tr>
<tr>
<td>Controller</td>
<td>Make the controller reusable (subclassable) and start database cursors automatically, open the Event Editor, and connect the interface to an external database. This menu appears on the menu bar when you create or edit an interface.</td>
</tr>
<tr>
<td>Component</td>
<td>Connect the interface to an external database and open the Event Editor. This menu appears on the menu bar when you create or edit an interface component.</td>
</tr>
<tr>
<td>Form</td>
<td>Create, edit, and delete forms; change the grid and enable or disable the snap feature; change the order of item traversal on the form; and define return key methods and accelerator keys.</td>
</tr>
<tr>
<td>FormItem</td>
<td>Create, edit, and align, form items, and work with custom controls.</td>
</tr>
<tr>
<td>Tools</td>
<td>Access other ObjectStudio tools, add or delete instance variables, create or edit menus, and define synchronized lists.</td>
</tr>
<tr>
<td>Options</td>
<td>Specify the display settings of the toolbar, alignment toolbar, item palette, and status line.</td>
</tr>
<tr>
<td>Help</td>
<td>Access online Help about the Designer.</td>
</tr>
</tbody>
</table>

See “Designer menu overview” on page 25 for descriptions of each menu option.

Toolbar

The Designer toolbar, shown in the following figure, provides shortcuts to some of the most commonly used menu options. For example, you can save a file by clicking Save on the toolbar.

Each button on the toolbar has an associated tooltip, which briefly describes the button’s function. To display the tooltip, place the cursor over the button and wait for the tooltip to display (make sure that the Designer window has focus).
Alignment toolbar
The Alignment toolbar provides shortcuts to the menu items on the FormItem > Align menu. As with the Designer toolbar, you can display tooltips to identify the alignment options.

For more information about the Alignment toolbar, see “Aligning form items” on page 133.

Status line
The status line displays expanded versions of the tooltips that are displayed for toolbar Buttons. Also, the status line displays hints about dialog boxes and class information about form items on a form. Here is an example of a status line:

Displaying the toolbars and status line
By default, the Designer displays a toolbar, an Alignment toolbar, and a status line. To remove or add these tools:

- Select Options > Show Toolbar from the Designer menu to display or hide the toolbar.
- Select Options > Show Alignment Toolbar from the Designer menu to display or hide the Alignment toolbar.
- Select Options > Show Status Line from the Designer menu to display or hide the status line.

When you select either of these menu items:

- If the tool is not displayed, the system displays the tool and places a check mark next to the option you selected.
- If the tool is displayed, the system removes the tool and removes the check mark next to the option you selected.
Designer modes

The Designer has two modes:

♦ **Edit Interface mode:** The Edit Interface mode is the default Designer mode. In this mode, all of the components described in the previous sections are available to help you create and edit interfaces.

To access the Edit Interface mode from the Test Interface mode, select **File > Edit Interface** from the menu bar.

♦ **Test Interface mode:** The Test Interface mode runs the interface currently open in the Designer. Because this mode simply tests the interface, only the menu bar appears. To access any other Designer components, you must switch back to the Edit Interface mode. Closing the interface in test mode automatically switches back to Edit mode.

To access the Test Interface mode, select **File > Test Interface** from the menu bar or click the **Test Interface** icon on the toolbar.
Exiting the Designer

To exit the Designer, select **File > Exit** from the Designer menu or double-click the **System Menu** in the upper-left corner of the Designer window.

If you are in Test Interface mode, you cannot exit the Designer. Return to Edit Interface mode to exit.

If you have made changes since you last saved your work, ObjectStudio prompts you to save your work before exiting.

The following table describes the Buttons you can click on the dialog box:

<table>
<thead>
<tr>
<th>If you click</th>
<th>Then the system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Displays the Save As dialog box the first time you save. After the first time, your work is saved to the last file you specified.</td>
</tr>
<tr>
<td>No</td>
<td>Does not save your work before exiting.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Resumes work in the Designer.</td>
</tr>
</tbody>
</table>

For more information about using the Save and Save As dialog boxes, see “Saving an interface” on page 72.
Designer menu overview

This section describes the menus available from the Designer menu bar. Each menu is described by:

- A table that briefly describes the menu items
- Where applicable, a pointer to the section of this manual where the menu item is explained in more detail

File menu

The following table describes the File menu, which allows you to save and test your work and exit the Designer:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save</td>
<td>Save your Designer work to the last file where you saved.</td>
<td>“Saving an interface” on page 72</td>
</tr>
<tr>
<td>Save as</td>
<td>Save your Designer work to a file that you specify.</td>
<td>“Saving an interface” on page 72</td>
</tr>
<tr>
<td>Test interface/Edit interface</td>
<td>Enter test mode and return to edit mode, respectively.</td>
<td>“Testing the interface” on page 289</td>
</tr>
<tr>
<td>Exit</td>
<td>Exit the Designer. Gives you the option of saving unsaved work.</td>
<td>“Exiting the Designer” on page 24</td>
</tr>
</tbody>
</table>
Controller menu

The following table describes the Controller menu, which displays on the menu bar only when you create or edit an interface:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Options</td>
<td>Make the controller reusable (subclassable) and start database cursors</td>
<td>“Making an interface into a template” on page 250</td>
</tr>
<tr>
<td></td>
<td>automatically.</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td>Open the Event Editor.</td>
<td>“General events” on page 317</td>
</tr>
<tr>
<td>External Database</td>
<td>Connect the interface to an external database.</td>
<td>ObjectStudio Database User’s Guide, P40-3803</td>
</tr>
</tbody>
</table>

Component menu

The following table describes the Component menu, which displays on the menu bar only when you create or edit an interface component:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Options</td>
<td>Make the interface component reusable (subclassable).</td>
<td>“Making an interface into a template” on page 250</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td>Open the Event Editor.</td>
<td>“General events” on page 317</td>
</tr>
</tbody>
</table>
Form menu

The following table describes the Form menu, which allows you to create and edit forms:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>New form</td>
<td>Create a new form.</td>
<td>“Creating a form” on page 45</td>
</tr>
<tr>
<td>New subform</td>
<td>Create a new subform.</td>
<td>“Creating a form” on page 45</td>
</tr>
<tr>
<td>New dialog box</td>
<td>Create a new dialog box.</td>
<td>“Creating a form” on page 45</td>
</tr>
<tr>
<td>New modal dialog box</td>
<td>Create a new modal dialog box.</td>
<td>“Creating a form” on page 45</td>
</tr>
<tr>
<td>New system modal dialog box</td>
<td>Create a new system modal dialog box.</td>
<td>“Creating a form” on page 45</td>
</tr>
<tr>
<td>New Property Page</td>
<td>Create a form that can be added to a property sheet.</td>
<td>“Creating property pages and property sheets” on page 114</td>
</tr>
<tr>
<td>Property Sheet</td>
<td>Create a container for property pages.</td>
<td>“Creating property pages and property sheets” on page 114</td>
</tr>
<tr>
<td>Clear form</td>
<td>Delete all items from the form that has focus.</td>
<td>“Clearing a form” on page 47</td>
</tr>
<tr>
<td>Delete form</td>
<td>Delete the form that has focus.</td>
<td>“Deleting a form” on page 48</td>
</tr>
<tr>
<td>Change form</td>
<td>Edit the form that has focus.</td>
<td>“Changing a form” on page 49</td>
</tr>
<tr>
<td>Grid and snap</td>
<td>Change the grid and enable or disable snap.</td>
<td>“Using the grid” on page 130</td>
</tr>
<tr>
<td>Item traversal</td>
<td>Change the order of item traversal on the form.</td>
<td>“Using the default tab order on a form or specifying your own” on page 55</td>
</tr>
<tr>
<td>Return key methods</td>
<td>Define return key methods.</td>
<td>“Assigning a return key method” on page 287</td>
</tr>
<tr>
<td>Accelerator keys</td>
<td>Define accelerator keys.</td>
<td>“Form events” on page 310</td>
</tr>
</tbody>
</table>

The bottom of the Form menu displays a list of open forms, subforms, or MDI forms. You can select a name from the list to change focus to the selected form.
### FormItem menu

The following table describes the FormItem menu, which allows you to create and edit form items:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Item</td>
<td>Display the New Item dialog box so that you can create form items.</td>
<td>“Creating a form item” on page 125</td>
</tr>
<tr>
<td>Change Options</td>
<td>Display the options dialog box for the selected form item or group of items.</td>
<td>“Customizing form items” on page 137</td>
</tr>
<tr>
<td>Change Class</td>
<td>Change the class of the selected form item.</td>
<td>“Changing the class of a form item” on page 136</td>
</tr>
<tr>
<td>Copy Item(s)</td>
<td>Copy the selected form item or items.</td>
<td>“Creating a form item” on page 125</td>
</tr>
<tr>
<td>Delete Item(s)</td>
<td>Delete the selected form item or items.</td>
<td>“Deleting a form item” on page 129</td>
</tr>
<tr>
<td>Deselect Item(s)</td>
<td>Deselect all selected form items.</td>
<td>“Selecting a form item” on page 129</td>
</tr>
<tr>
<td>Select all</td>
<td>Select all form items on the form that has focus.</td>
<td>“Selecting a form item” on page 129</td>
</tr>
<tr>
<td>Align</td>
<td>Perform the same functions as the Alignment toolbar.</td>
<td>“Aligning form items” on page 133</td>
</tr>
<tr>
<td>Custom Control</td>
<td>Display the Property Editor (Ambient Controls) or register an OCX file (Register Control).</td>
<td>ObjectStudio OLE User’s Guide, P40-3805</td>
</tr>
</tbody>
</table>
Tools menu

The following table describes the Tools menu, which gives you access to other ObjectStudio tools from within the Designer:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>Display the Method Editor.</td>
<td>&quot;Using the Method Editor to create a method&quot; on page 61</td>
</tr>
<tr>
<td>Variables</td>
<td>Add or delete instance variables.</td>
<td>&quot;Working with instance variables&quot; on page 270</td>
</tr>
<tr>
<td>Menus</td>
<td>Create or edit menus.</td>
<td>&quot;Menu editor&quot; on page 75</td>
</tr>
<tr>
<td>Toolbar</td>
<td>Create or edit toolbars.</td>
<td>&quot;Toolbar editor&quot; on page 98</td>
</tr>
<tr>
<td>Status Line</td>
<td>Create or edit status lines.</td>
<td>&quot;Status line editor&quot; on page 107</td>
</tr>
<tr>
<td>Tooltip</td>
<td>Create tooltips for Buttons on a toolbar.</td>
<td>&quot;Tooltip editor&quot; on page 113</td>
</tr>
<tr>
<td>Synchronized</td>
<td>Define synchronized lists.</td>
<td>&quot;List Box&quot; on page 190</td>
</tr>
<tr>
<td>SQL Editor</td>
<td>Display the SQL Editor.</td>
<td>ObjectStudio Database User's Guide, P40-3803</td>
</tr>
</tbody>
</table>
### Options menu

The following table describes the Options menu, which allows you to toggle the display of Designer tools:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Toolbar</td>
<td>Toggle the display of the Designer toolbar.</td>
<td>“Toolbar” on page 21</td>
</tr>
<tr>
<td>Show Alignment Toolbar</td>
<td>Toggle the display of the Alignment toolbar.</td>
<td>“Alignment toolbar” on page 22</td>
</tr>
<tr>
<td>Show Status Line</td>
<td>Toggle the display of the status line.</td>
<td>“Status line” on page 22</td>
</tr>
</tbody>
</table>

### Help menu

The following table describes the Help menu, which gives you access to online Help about the Designer:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help for help</td>
<td>Describes how to use the Help system.</td>
</tr>
<tr>
<td>Extended help</td>
<td>Displays the top-level Help page for the Designer.</td>
</tr>
<tr>
<td>Help index</td>
<td>Opens ObjectStudio Help.</td>
</tr>
<tr>
<td>About</td>
<td>Displays a product overview.</td>
</tr>
</tbody>
</table>
Quick tour

This tutorial introduces you to the most basic features of the Designer. In general, the tutorial presents one way to accomplish a task, even when there are multiple ways to do the same thing. Later chapters present more information for specific tasks.

**What you will build.** During this tutorial, you create a small application that includes the dialog box shown here:

As you go through the tutorial, use this picture as a guide when you create and arrange the items that belong in the dialog box.

**To use the application:** To use the finished application, type your name in the Entry Field. When you click **Greet**, the system displays a message box that greets you by name. When you click **Done**, the system closes the dialog box.
Stages to create the application. To create the application, the tutorial leads you through the following sections:

♦ “Starting the Designer” on page 32
♦ “Creating the Static Text item” on page 34
♦ “Creating the Entry Field item” on page 35
♦ “Creating the Greet button” on page 36
♦ “Creating the Done button” on page 37
♦ “Creating the greetMe method” on page 39
♦ “Linking the method to a Button” on page 41
♦ “Saving changes” on page 42
♦ “Testing the interface” on page 43

The following sections present each step in detail.

Starting the Designer

To start the tutorial:

   The ObjectStudio Desktop appears.

2. Click the Create New Interface button (shown at left) on the Desktop toolbar.
   The Select Controller Type dialog box appears.

3. Click OK on the Select Controller Type dialog box (accept the default controller).
   The Designer appears, with a new form entitled “Test”.

4. Resize the Test form to about half of its original height.
   To resize a form in the Designer, use the standard Windows procedure for resizing a window.
5. Select **FormItem > New Item** from the Designer menu.

The New Item dialog box appears:

Some items pictured here will be missing if you have not loaded the OLE loadable application.

This dialog box contains a Value Set, or graphical array, of items that you can add to the form.

As you click each item, text displays under the **Name** entry field, telling you the kind of item you have selected.
You can now add items to the form using one of the following methods:

- Select the form item and drag it to the form.
- Select the form item and press ENTER.
- Double-click the form item (the form item is added to the form automatically).

You can name the form item before you add it to the form (type a name in the Name entry field) or you can name it after you add it to the form. As you work with the Designer, you will discover the style that is best for you. For more information about creating form items, see “3. Creating form items” on page 120.

Creating the Static Text item

To create the Static Text item:

1. Select the Static Text icon in the New Item dialog box.
   The system highlights the Static Text icon and displays the words “Static Text” under the Name entry field.

2. Place the cursor in the Name entry field and type:
   Please enter your name:

3. Drag the Static Text icon to the form. (Remember to use mouse button 1 to drag.)

4. To move the item after you place it, click it and then drag it with mouse button 1.

The form now should look like the one shown here:
Creating the Entry Field item

To create the Entry Field item:

1. Double-click the **Entry Field** icon in the New Item dialog box.
   
   A new Entry Field item displays on the form.

2. Move the Entry Field item until it is below the Static Text item you just created.

3. Resize the Entry Field item to twice its original width. To resize an item:
   
   A. Select the item.
   
   B. Use mouse button 1 to drag one of the item’s selection handles until the item is the size you want.

The form now should look like the one shown here:
Creating the Greet button

To create the Greet button:

1. Type the following in the Name entry field of the New Item dialog box:
   Greet
2. Select and drag the Button icon to the lower-left corner of the form.

The form now should look like the one shown here:

![Example form with a Greet button](image)

“Creating the greetMe method” on page 39 shows you how to write a method that executes when the user clicks Greet.
Creating the Done button

To create the Done button:

1. Drag the Button icon from the New Item dialog box to the lower-right corner of the form.

2. Double-click the Button form item on the form.

   The Form Button Options dialog box for the new Button displays, as shown here:

   ![Form Button Options dialog box]

   - Name: button1
   - Text: button1
   - Button Type: Text
   - Link: [Link...]

3. Press Tab once to place the cursor in the Text entry field.

   For now, you can ignore the Name entry field. To learn more about its importance, see “Working with instance variables” on page 270.

4. Type the following in the Text entry field:

   Done

   Do not close the Form Button Options dialog box.
**Associating an item with an activity**

To associate a predefined activity with an item on a form:

1. Click the arrow of the **Custom** drop-down list box.
   
   The choices display in the Drop-down List Box. These choices represent the predefined activities that you can associate with an item on a form.

2. Select the **Close** option.

3. Click **OK**.

   The Button is now labeled “Done” and is associated with the close activity.

The form now should look like the one shown here:

![Image of a form with buttons labeled Greet and Done]
Creating the greetMe method

In this section, you will learn how to create a method within the Designer. Later, you will learn how to link the method to an item on the form.

To open the Method Editor from the Designer:

1. Select **Tools > Methods** from the Designer menu.

The Method Editor displays, as shown here:
2. Click **New**.
   The system places the cursor in the **Source** edit area. You are now ready to enter code for the method.

3. Type the following lines of code in the **Source** edit area:
   ```
greetMe
    self infoBoxMessage:
        'Hi' ++ (entryField1 getValue) + '!'.
   ```

4. Click **Save**.

5. Click **Close**.
   The system is now ready to use the method. You must link the method to the Greet button.

**What the method does:** The `greetMe` method displays a message box that gets the name typed into the Entry Field of the form and displays a greeting. If the user types Mr. Flanagan into the Entry Field, the corresponding message box displays, as shown here:

![Information Test](image)
Linking the method to a Button

Now that you have created the greetMe method, you need to link it to the Greet button. After you do so, when the user clicks Greet, the greetMe method executes.

To link the method to the Button:

1. Double-click Greet.
   The Form Button Options dialog box displays.
2. Click Methods.
   The Method Assignment dialog box displays, as shown here:
3. Select `greetMe` from the Available Methods list box.

4. Click Copy.
   
The `greetMe` method displays in the Assigned Methods list box.

5. Click OK to close the Method Assignment dialog box.
   
   Focus returns to the Form Button Options dialog box.

6. Click OK to close the Form Button Options dialog box.

   **You are done!** You have finished creating the Greet Me application.

   Check your work to make sure that it looks like the interface shown in the figure in “Quick tour” on page 31.

**Saving changes**

This is a good time to save the application.

To save the application:

1. Select `File` > `Save as` from the Designer menu.

2. Select the folder in which you want to save the file from the Save in drop-down list box.

3. Type a file name in the File name entry field. For example, type `greetme.cls`. The default extension for a class file is `.cls`.

4. Click Save to save the file.
Testing the interface

Test the interface to ensure that you performed all steps correctly.

To test the interface:

1. Select File > Test interface from the Designer menu.
   
   The system displays the form that you just defined. In addition, the 
   Designer’s window changes:

   A. The title changes to Designer – Test Mode.
   
   B. All functionality in the Designer is disabled except for the File menu 
      and the Test interface button on the toolbar.

2. Type your name in the Entry Field and click Greet.
   
   The greeting message box displays.

3. Click OK to close the message box.

4. Click Done to close the Greet dialog box.
If something goes wrong: If the test does not work correctly, you might see a Notifier dialog, similar to the one shown here:

![Exception dialog](image)

Unhandled exception: Message not understood: #gtValue

ObjectStudio.StringCtrl(...ct)>>doesNotUnderstand:
ObjectStudio.StringCtrl(...doesNotUnderstand:args:
ObjectStudio.StringCtrl(...ct)>>doesNotUnderstand:
ObjectStudio.TestController>>greetMe
ObjectStudio.TestController(Object)>>os_perform:

For now:

1. Click **Terminate**.
   
The Exception dialog closes.

2. Select **File > Edit interface** from the Designer menu.
   
The system returns to the Designer environment.

Try to figure out what went wrong. Start by comparing the code for the greetMe method to the code in this section. When you find the problem, fix it, save the interface, and retest.

If it all works: If everything works as planned, select **File > Edit interface** from the Designer menu to return to Edit Interface mode in the Designer.

You have finished working on the tutorial. You can:

- Continue to explore the Designer on your own.
- Exit the Designer for now. For help with this step, see “Exiting the Designer” on page 24.
2. Creating forms

Working with forms

This section shows you how to work with forms.

Creating a form

There are two ways to create a form in the Designer:

♦ As a side effect of creating a new interface, the system creates the main form. For more information on the main form, see “Form” on page 14.

♦ You can create additional forms by selecting options from the Form pull-down menu.
Recall that each form you create depends on the main form. The following table describes the forms you can create by selecting an option from the Form pull-down menu:

<table>
<thead>
<tr>
<th>Select</th>
<th>To create</th>
</tr>
</thead>
<tbody>
<tr>
<td>New form</td>
<td>A form that looks like the main form. By default, it can have a title bar and menu bar. It can be resized, and it can be moved independently of the main form.</td>
</tr>
<tr>
<td>New subform</td>
<td>A child window of the form that currently has focus. By default, a subform can have a title bar, toolbars, and status lines, and it can be resized. It can be moved but is clipped by the main form.</td>
</tr>
<tr>
<td>New dialog box</td>
<td>A specialized version of a new form. A dialog box cannot be minimized or maximized, and it has a special border.</td>
</tr>
<tr>
<td>New modal dialog box</td>
<td>A specialized version of a dialog box. You must interact with the dialog box and close it before you can interact with the rest of the application.</td>
</tr>
<tr>
<td>New system modal dialog box</td>
<td>A specialized version of a dialog box. You must interact with the dialog box and close it before you can interact with other applications or the operating system. On multitasking systems, you can interact with other applications and the operating system; however, the dialog box remains on top until you close it.</td>
</tr>
<tr>
<td>New property page</td>
<td>A form (page) that is added to a property sheet. Each form becomes a tabbed page on the property sheet.</td>
</tr>
<tr>
<td>Property sheet</td>
<td>A container for property pages.</td>
</tr>
</tbody>
</table>

**MDI forms:** You also can create Multiple Document Interface (MDI) forms. These forms are native Windows forms (MDIController, whose forms are MDIWindows and MDIWindowsChildForm). The behavior of these forms is different from that of the forms described in the previous table. For more information about MDI forms, see “Multiple document interface” on page 272.
Setting focus to a form

To set focus to a form, select Form > Form name from the Designer menu. Alternatively, click the title bar of the form.

To find out which form has focus, display the Form pull-down menu. The form name that has a check mark next to it has focus.

Clearing a form

When you clear a form, the system destroys all form items that belong to the form. However, it leaves any changes that you made with the Form > Change form option on the Designer menu.

To clear a form:

1. Set focus to the form that you want to clear.
2. Select Form > Clear form from the Designer menu.

The system clears the form.
Deleting a form

When you delete a form, the system destroys the form and its contents. It removes the form from the Designer and generates no Smalltalk code for the form.

To delete a form:
1. Set focus to the form that you want to delete.
2. Select Form > Delete form from the Designer menu.
   The system asks you to confirm the deletion.
3. Click Yes to delete the form or click No to cancel the deletion.

Deleting the main form

If you delete the main form in the interface and the interface contains other forms that are not subforms of the main form, the system selects another form as the new main form.

When you cannot delete a form

The system prevents you from deleting a form when it is:

♦ The only form in the interface
♦ The only main form in the interface
Changing a form

The Designer allows you to change visual and behavioral characteristics of a form by working with the Change Form dialog box.

To display the Change Form dialog box:
1. Set focus to the form you want to edit.
2. Select Form > Change form from the Designer menu.

The Change Form dialog box displays, as shown here:

Exiting the Change Form dialog box

To exit the dialog box, click OK to save your changes or click Cancel to exit without making any changes.
Dialog box settings

The settings on the Change Form dialog box are:

- **Form Name**: The Form Name is the name of the instance of class Form that the system creates when you save the interface. By default, the system generates the form name.

- **Form Title**: The Form Title is the text that displays in the form's title bar. By default, the form title is the same as the form name.
  
  For information about how to name an interface, see “Naming an interface” on page 71.

- **Initial Focus**: The Initial Focus drop-down list box lists the form items in the form. Select the form item that should receive focus when the form opens. By default, the item with initial focus is the first item you create.

- **Pop-up Menu**: The Pop-up Menu drop-down list box lists all menus you have created for which you checked the Not in Menu Bar check box. Select the menu that you want to make the form's pop-up menu. By default, there is no pop-up menu.
  
  To learn more about creating pop-up menus, see “Creating a pop-up menu” on page 90.
**Options:** The following table describes the effect of checking the items in the Options topic box on the Change Form dialog box. Items in the table are listed alphabetically.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning when checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add to Task List</td>
<td>The form is added to the system task list so that the user can switch to the form.</td>
</tr>
<tr>
<td>Dialog Border</td>
<td>The form has a border that does not allow resizing the form. When you check this option, checking Sizing Border and Solid Border has no effect. In Windows, you cannot check this option for a form without a title bar.</td>
</tr>
<tr>
<td>Horz. Scroll Bar</td>
<td>The form has a horizontal scroll bar.</td>
</tr>
<tr>
<td>Maximized</td>
<td>The window is maximized when it opens. By default, when the window opens, its state is restored. If you check this option, you must leave Minimized unchecked.</td>
</tr>
<tr>
<td>Min Max Button</td>
<td>The form has Minimize and Maximize buttons in the upper-right corner, and the Minimize and Maximize choices are enabled on the System Menu. If you check this option, the form must have a title bar. (Checked by default.)</td>
</tr>
<tr>
<td>Minimized</td>
<td>The window is minimized when it opens. By default, when the window opens, its state is restored. If you check this option, you must leave Maximized unchecked.</td>
</tr>
<tr>
<td>Never Active</td>
<td>The form does not accept focus when the user clicks in the form. Instead, the main form gets focus.</td>
</tr>
<tr>
<td>Not In Windows Menu</td>
<td>The name of the form does not display in the Window menu.</td>
</tr>
<tr>
<td>Sizing Border</td>
<td>The form has a border that allows resizing the form. If you check this option, you must leave Dialog Border and Solid Border unchecked. (Checked by default.)</td>
</tr>
<tr>
<td>Solid Border</td>
<td>The form has a solid border that does not allow resizing the form. If you check this option, you must leave Dialog Border and Sizing Border unchecked.</td>
</tr>
<tr>
<td>System Menu</td>
<td>The form has a System Menu (the area at the top-left corner of the window). If you check this option, the form must have a title bar. (Checked by default.)</td>
</tr>
<tr>
<td>Title Bar</td>
<td>This option if valid for subforms only. When selected, the title bar will be displayed.</td>
</tr>
<tr>
<td>Vert. Scroll Bar</td>
<td>The form has a vertical scroll bar.</td>
</tr>
</tbody>
</table>
MDI forms have horizontal and vertical scroll bars by default. The scroll bars are managed by Windows. The client width and client height are the same as the width and height of the form's rectangle (without the title bar).

If the form is smaller than the area over which the form can scroll, and a form item outside the displayed area is given focus, the form can automatically scroll to display the form item that has been given focus. For this to occur, the form must have scrollbar(s) enabled and the form item must be completely hidden (if it is partially visible, automatic scrolling will not occur). Like a form, a Notebook page can automatically scroll to display a hidden item that is given focus while the Notebook page is not completely displayed on its parent form.

♦ **Client width and client height:** These options apply only when the form has scroll bars:
  - **Client Width** is valid only when the form has a horizontal scroll bar
  - **Client Height** is valid only when the form has a vertical scroll bar

**Client Width** and **Client Height** declare, in tenths of millimeters, the height or width of the area over which the form can scroll. Imagine a form as a scrolling view into a large area.

The large area is defined by **Client Width** and **Client Height**, as shown here:
**Background**: The Designer allows you to change the background of a form to a bitmap, a predefined color, or a color that you define. To change the background of a form, click **Background** on the Change Form dialog box.

The Form Background Options dialog box displays, as shown here:

To set the background of a form to a bitmap:

1. Select the **Bitmap** radio button on the Form Background Options dialog box.
2. Type the absolute or relative pathname of the bitmap file in the **Bitmap File Specification** entry field.
3. Select one of the following from the **Bitmap Orientation** box:
   - **Fill**, to fill the form with one enlarged bitmap
   - **Tile**, to fill the form by repeating the bitmap pattern
   - **Center**, to place a bitmap in the center of the form
4. Click **OK** to save your changes or click **Cancel** to exit without saving.
To set the background of a form to a color:

1. Select the **Color** radio button on the Form Background Options dialog box.
   
The **Selected Color** drop-down list box displays on the dialog box.

2. Select the color that you want to assign to the form from the **Selected Color** drop-down list box.
   
The color of the example box changes to reflect your choice.

3. Click **OK** to save your changes or click **Cancel** to exit without saving.

To set the background of a form to a custom color (RGB):

1. Select the **RGB** radio button on the Form Background Options dialog box.
   
   Three Sliders that are labeled **Red**, **Green**, and **Blue** display on the dialog box.

2. Adjust the Sliders for the background color you want.
   
   The color of the example box changes as you move each Slider.

3. Click **OK** to save your changes or click **Cancel** to exit without saving.
Using the default tab order on a form or specifying your own

Introduction

When a form has focus, usually one form item on that form also has focus. If the user presses the Tab key, focus moves to the next form item in the tab order. “Tab order” specifies to which form items, and in what order, the user can tab. Perform either of the following:

♦ Use the default tab order.
♦ Manually define a new tab order.

After you are satisfied with the tab order, you can test it.

Using the default tab order

When you add form items to a form item, ObjectStudio automatically generates a default tab order that reflects the order in which you add form items.
Manually defining a new tab order
You may wish to manually define the tab order so that it flows from left to right and from the top of the form to the bottom.

Rearranging items within the tab order: To rearrange items within the tab order, perform the following:
1. On the Designer, select the Form > Item traversal menu option.
   The Item Groups dialog box appears, as shown here:

   ![Item Groups dialog box]

   The In Group list box displays an ordered list of all items to which you can tab.

   2. Use the right mouse button to drag the item, in the In Group list box, whose tab order you wish to change.

   As you drag the item, a horizontal line appears between two items on the list. The line moves as you drag. When you release the mouse button, the item you are moving is inserted at the location of the horizontal line.
Excluding an item from the tab order: To prevent the user from tabbing to an item, perform the following:

1. Select the item, in the In Group list box, that you wish to exclude.
2. Click the Exclude button.

The selected item moves from the In Group list box to the Not In Group list box.

Including an excluded item: To allow the user to tab to an item that is currently excluded from the tab order, perform the following:

1. Select an item in the Not In Group list box.
2. Click the Include button.

The selected item moves from the Not In Group list box to the end of the In Group list box.

Grouping items within the tab order: You can group items so that they are treated as a unit by the tab order. When the user tabs to any item in the group and presses the Tab key again, focus goes to the next item that is not in the group. The user can navigate within a group by pressing the arrow keys.

For example, you may wish to group items that are inside a Topic Box.

To group a set of items, perform the following:

1. Select multiple items in the In Group list box.
2. Click the Group button.

ObjectStudio replaces the selected items with a single item. The new item is an array that contains the selected items.

(Indirectly) setting a mnemonic on an Entry Field: This is useful when you wish to enable an application user to quickly set focus to an Entry Field. First, set a mnemonic on a Static Text Field by placing an ampersand (&) character in the label text, right before the letter you wish to use as the underlined mnemonic letter. Next, use the Item Groups dialog box (opened from the Designer’s Form > Item traversal menu option) to group the Static Text Field with an Entry Field (make sure that the Static Text Field is the first form item in the group and that the Entry Field is the second). Now, when the user selects the mnemonic using Alt+<the underlined letter>, focus goes straight to the grouped Entry Field. For more information on mnemonics, see “Creating mnemonics for form items” on page 139.
**Ungrouping items:** To ungroup a group of items, perform the following:

1. Select a group of items in the In Group list box.
2. Click the Degroup button.

ObjectStudio replaces the group with the individual items that are in the group.

**Exiting the Item Groups dialog box:** To exit the dialog box, perform one of the following:

- **Click OK.** This saves your changes.
- **Click Cancel.** This exits the dialog box without saving your changes.

**Testing the tab order**

When you change tab order, you may not be able to see the results in the Designer. To view the results, perform either of the following:

- **Test the interface.** To do so, select the File > Test interface menu option from the Designer.
- **Save the interface and open it from the Desktop.**
Modal dialog boxes

ObjectStudio uses native Windows modality. The only type of form that can be modal is a dialog box.

Types of modal dialog boxes

There are two types of modality:

♦ Application modal (modal): A modal dialog box prohibits the user from interacting with the application until the dialog box closes.

♦ System modal: A system modal dialog box allows the user to interact with forms that are not its parent, but the modal dialog box remains on top until it closes.

Both types of dialog boxes are modal to their parent form and prohibit the user from interacting with the parent until the dialog box closes. For example, all Form Options dialog boxes are modal to the Designer. You cannot interact with the Designer until you close the Form Options dialog box.

All modal dialog boxes must have a parent. If you do not want to have a parent form, use a nonmodal dialog box.
Synchronism

Modal dialog boxes are synchronous. Once they are open, all other Smalltalk code pauses until the dialog box closes. This synchronization means that you can write methods that send the open message to a modal dialog box, and you can then send messages immediately to gather values of each field in the modal dialog box. The code that gathers values does not execute until the modal dialog box closes.

Modal message boxes

Message boxes, which are subclasses of dialog boxes, also can be modal. ObjectStudio supports message boxes that can be modeless or modal. There are two types of message boxes:

♦ MessageBox, which is a subclass of Form
♦ ModalMessageBox, which is a subclass of ModalDialogBox

Creating a modal dialog box

To create a modal dialog box:

1. Click New Interface on the Desktop toolbar to create a new interface and to open the Designer.

2. From the Designer menu, select Form > New modal dialog box or Form > New system modal dialog box.

When you are creating a modal dialog box, keep in mind that modal dialog boxes do not support menus, toolbars, or status lines.
Using the Method Editor to create a method

The Designer allows you to use the Method Editor to define methods.

Opening the Method Editor

To open the Method Editor, select Tools > Methods from the Designer menu. The Method Editor displays, as shown here:

The rest of this section describes the Method Editor.
**Method Editor buttons**

The following table describes the Buttons on the Method Editor:

<table>
<thead>
<tr>
<th>Click</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Create a new method</td>
</tr>
<tr>
<td>Save</td>
<td>Save the current method</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete the current method</td>
</tr>
<tr>
<td>Close</td>
<td>Close the dialog box</td>
</tr>
<tr>
<td>Undo</td>
<td>Delete any text entered in the Source edit area</td>
</tr>
</tbody>
</table>

**Method Editor pop-up menu**

The Method Editor has a Text Editor pop-up menu that provides easy access to commonly used functions. To use it, press mouse button 2 in the Source edit area while the Method Editor has focus.

For more information about the Text Editor, refer to the *ObjectStudio User’s Guide*, P40-3807.
Creating and editing methods in the Method Editor

The Method Editor allows you to create a new method or edit an existing method.

Creating a new method

To create a new method:

1. Select the Source radio button.
2. Select the Instance or Class radio button to reflect the type of method that you are creating.
3. Click New.
4. Type the method source code in the Source edit area.
5. Click Save to save the method. Click Delete if you do not want to save the method.
6. Click Close to close the Method Editor.

ObjectStudio uses the label you enter in the first line of the method as the name of the method when you click Save.

Editing an existing method

To edit an existing method, select the method that you want to edit from the Methods drop-down list box. The Method Editor displays the method in the Source edit area.
Method Editor shortcut list boxes and buttons

The Method Editor provides shortcut buttons for entering text, as described in the following table:

<table>
<thead>
<tr>
<th>To insert</th>
<th>Do the following</th>
</tr>
</thead>
<tbody>
<tr>
<td>An object name</td>
<td>1. Select the object to insert from the Object Insert list box.</td>
</tr>
<tr>
<td></td>
<td>2. Press the arrow button (&lt;) under the Object Insert list box.</td>
</tr>
<tr>
<td>A method name</td>
<td>1. Select the method to insert from the Method Insert drop-down list box.</td>
</tr>
<tr>
<td></td>
<td>2. Press the arrow button (&lt;) under the Method Insert list box.</td>
</tr>
<tr>
<td>A receiver object name and the message it receives</td>
<td>1. Select the object to insert from the Object Insert list box.</td>
</tr>
<tr>
<td></td>
<td>2. Select the method to insert from the Method Insert list box.</td>
</tr>
<tr>
<td></td>
<td>3. Press the double-arrow button (&lt;&lt;) under the Method Insert list box.</td>
</tr>
</tbody>
</table>
Automation Function Helper

This feature of the Method Editor generates Smalltalk code fragments for ActiveX controls (referred to in ObjectStudio and much of its documentation, including the following documentation, as “OLE Custom Controls”). For experienced users, brief instructions are provided below. For less-experienced users, detailed instructions are provided under “Detailed Automation Function Helper instructions” on page 66.

Brief Automation Function Helper instructions

Opening the Automation Function Helper: To open the Automation Function Helper, perform the following:

1. Create an OLE Custom Control.
2. Open the Method Editor.
3. Select the newly created OLE Custom Control from the Object Insert list box.
4. Select <Automation Function> from the Method Insert list box.
5. Click either the arrow button (>) or the double-arrow button (<<).

The Automation Function Helper appears.

Using the Automation Function Helper: To use the Automation Function Helper, perform either of the following:

♦ Generate code to PUT or GET an OLE property of the OLE Custom Control: Make sure that the toggle button below the left-hand list box displays the word Methods, select the desired OLE property from the scrolling list, select either the Property GET or Property PUT radio button, and click OK.

♦ Generate code to call an OLE method of the OLE Custom Control: Make sure that the toggle button below the left-hand list box displays the word Properties, select the desired OLE method from the scrolling list, and click OK.

The Automation Function Helper closes and ObjectStudio pastes the generated Smalltalk code fragment into the Source edit area of the Method Editor.

For more detailed instructions, see “Detailed Automation Function Helper instructions” on page 66.
Detailed Automation Function Helper instructions

Preparing to use the Automation Function Helper: To prepare to use the Automation Function Helper, perform the following:

1. **Load the OLE application.** To do so, perform the following:
   A. On the ObjectStudio desktop, select the File > Load application menu option.
      The Applications dialog box appears.
   B. Scroll down to OLE.
   C. Double-click OLE.
      ObjectStudio loads support for OLE.

2. **Create a new interface.** To do so, perform the following:
   A. On the ObjectStudio desktop, select the File > New > Interface menu option.
      The Select Controller Type dialog box appears.
   B. Click OK.
      The new interface window and the Designer appear.

3. **Add an OLE Custom Control form item to the new interface.** To do so, perform the following:
   A. On the Designer menu, select the FormItem > New Item menu option.
      The New Item dialog box appears.
   B. Double-click the OLE Custom Control icon.
      “OLE Custom Control” is an old name for an ActiveX control.
      The “Insert an OLE Control” dialog box appears. It could take up to a minute for the scrolling list to populate.
   C. Scroll down to the desired control type.
   D. Double-click the desired control type.
      A rectangle, containing the control, appears on your interface.

4. **Open the Method Editor.** To do so, on the Designer, select the Tools > Methods menu option.
   The Method Editor appears.
**Opening the Automation Function Helper.** To open the Automation Function Helper, perform the following:

1. In the Method Editor, select the new OLE Custom Control’s name from the **Object Insert** list box (located on the right-hand side of the Method Editor).

   `<Automation Function>` appears on the first line of the **Method Insert** list box.

2. Select `<Automation Function>` from the **Method Insert** list box.

3. Click one of the following:

   - **Arrow button (<):** Normally, clicking the arrow button immediately inserts the selected method name into the **Source** edit area of the Method Editor. However, since the receiver object that you selected is an OLE Custom Control, and the method that you selected is `<Automation Function>`, the Automation Function Helper appears.

   - **Double-arrow button (<<):** Normally, clicking the double-arrow button immediately inserts the selected receiver object name and selected method name into the **Source** edit area of the Method Editor. However, since the receiver object that you selected is an OLE Custom Control, and the method that you selected is `<Automation Function>`, the Automation Function Helper appears.

The Automation Function Helper appears.
Using the Automation Function Helper. To use the Automation Function Helper, perform either of the following:

- **Generate code to PUT or GET a property of the OLE Custom Control.** To do so, perform the following:

  1. If the toggle button below the left-hand list box displays the word **Properties**, click it.

     The Automation Function Helper toggles from **Methods to Properties** mode and now lists the names of all the OLE properties found in the selected OLE Custom Control.

  2. Select the desired **OLE** property from the left-hand list box.

     The Automation Function Helper displays the following:

     - In the **Description** area of the Automation Function Helper: A description of the selected OLE property.

     - At the bottom of the Automation Function Helper: The Smalltalk code fragment that will be generated in order to GET or PUT the selected OLE property.

  3. Choose to generate either a GET request or a PUT request for the selected OLE property by selecting the **Property GET** or **Property PUT** radio button.

  4. Click **OK**.

     The Automation Function Helper closes and ObjectStudio pastes the generated Smalltalk code fragment into the **Source** edit area of the Method Editor.
Generate code to call a method of the OLE Custom Control. To do so, perform the following:

1. If the toggle button below the left-hand list box displays **Methods**, click it.

   The Automation Function Helper toggles from Properties to Methods mode and now lists the names of all the OLE methods found in the selected OLE Custom Control.

2. Select the desired OLE method from the left-hand list box.

   The Automation Function Helper displays the following:
   - In the **Description** area of the Automation Function Helper: A description of the selected OLE method.
   - At the bottom of the Automation Function Helper: The Smalltalk code fragment that will be generated in order to call the selected OLE method.

3. Click **OK**.

   The Automation Function Helper closes and ObjectStudio pastes the generated Smalltalk code fragment into the **Source** edit area of the Method Editor.
Creating a method reference in the Method Editor

To create a method reference in the Method Editor, perform the following:

1. Create a new method or select a method from the Methods drop-down list box.
2. Select the Reference radio button.
3. Click New.
   - A reference template displays in the Source edit area.
4. Type reference text.
5. Click Save to save the method reference.
6. Click Close to close the Method Editor.
Naming an interface

It is useful to give the interface a descriptive name that identifies the icon when you open the interface. You can use the Controller Options dialog box to enter a new name for an interface.

To change the name that displays on the main form of the interface, you need to use the Form > Change form menu option.

To name an interface:

1. Select Controller > Change Options from the Designer menu.

The Controller Options dialog box displays, as shown here:

2. Type a name in the Controller Name entry field. For example, enter GreetMeDemo.

3. Click OK.
Saving an interface

There are two ways to save an interface:

♦ **Save.** Use the Save option to save without changing the file name. You usually keep the same file name when you are adding new information to an interface that you have already saved.

To save an interface, select **File > Save** from the Designer menu. If you have already assigned a file name to the interface, the system saves the interface and displays a confirmation message in the status line.

If you are saving an interface for the first time and select **File > Save** from the Designer menu, the system displays the Save As dialog box.

♦ **Save as.** Use the Save as option to save to a new file name. This option will keep the original file intact. The system displays this dialog box the first time when you save an interface or when you select the Save as option.

To save to a new file:

1. Select **File > Save as** from the Designer menu.

The Save As dialog box displays, as shown here:

![Save As dialog box](image)
2. Select the folder in which you want to save the file from the **Save in** drop-down list box.

   The **Keep Dir** check box remembers the folder you choose as the default location for class files for the remainder of this session. The next time you open the Save As dialog box, the same folder displays. If you do not want to use this folder as a default location for class files, uncheck the **Keep Dir** check box.

3. Select a file type from the **Save as type** drop-down list box. The default extension for a class file is .cls.

4. Type a file name in the **File name** entry field.

5. Click **Save** to save the changes or click **Cancel** to exit without making any changes.
Testing the interface

The Designer provides a testing tool that allows you to test an interface without having to exit the Designer.

To test the interface, select File > Test interface from the Designer menu.

It is recommended that you save the interface before you test it. For more information about saving the interface, see “Saving an interface” on page 72.

By default, the system displays the main form of the interface that you just defined. Also, the Designer’s window changes:

♦ The title changes to Designer – Test Mode
♦ All functionality in the Designer is disabled except for the File menu and the Test interface button on the toolbar

Now you can test the various components of the interface. For example, you can display other forms, click Buttons, or display menu items. Note that ObjectStudio reloads the class file when you enter Test Interface mode.

If something goes wrong. If the test does not work correctly, you might see a Notifier dialog, like the one shown here:

You can open the Debugger from the Notifier dialog by clicking the Debug button. For more information about the Debugger, refer to the Cincom Smalltalk Application Developer’s Guide, P46-0101.

Returning to the Designer: When you are ready to return to Edit Interface mode, select File > Edit interface from the Designer menu. The system returns to the Designer environment. Note that you must return to Edit Interface mode to exit the Designer.
Menu editor

A menu bar is the horizontal area that displays under the title bar of a window. The Designer allows you to create:

♦ Pull-down menus, which display in the menu bar
♦ Cascading submenus, which display in pull-down menus
♦ Pop-up menus, which display on the Desktop

Windows does not allow a subform to have a menu bar.
Getting started

To create a menu, start by displaying the Menu Setup dialog box. Select Tools > Menus from the Designer menu. The Menu Setup dialog box displays, as shown here:

Use the following areas of the Menu Setup dialog box to create a menu bar:

- Use the Menu Item Options group box to define individual elements of the menu.
- Use the vertical row of Buttons to move information about individual menu elements in the editor.
- Use the Menu Definition list box to view the menu hierarchy and make changes to the hierarchy.
- Use the Menu Options group box to add a standard Window menu and a standard Help menu to the menu bar. Also, you can specify the location of a Help file.
Adding items to the Menu Definition list box

To add items to the Menu Definition list box, click one of the vertical Buttons as described in the following table:

<table>
<thead>
<tr>
<th>Click</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Add the item in the Menu Item Options group box to the Menu Definition list box.</td>
</tr>
<tr>
<td>Insert</td>
<td>Insert the item in the Menu Item Options group box before the item in the Menu Definition list box that is selected.</td>
</tr>
<tr>
<td>Replace</td>
<td>Change the item in the Menu Definition list box that is selected, assigning it the attributes in the Menu Item Options group box.</td>
</tr>
<tr>
<td>Remove</td>
<td>Delete the item in the Menu Definition list box that is selected.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clear the Menu Item Options group box.</td>
</tr>
</tbody>
</table>
Adding menus to MDI forms

When you create an MDI form, it has a default menu bar that includes a File, Window, and Help menu. The Menu Setup dialog box displays, as shown here:

For more information about MDI forms, see “Multiple document interface” on page 272.

Exiting the Menu Setup dialog box

To exit the dialog box, click OK to save your changes or click Cancel to exit without making any changes.
Creating a menu bar

A typical menu bar might contain at least the File, Edit, Window, and Help menus, as shown in the following figure. These menus are pull-down menus that contain menu options and also can contain cascading menus.

![Menu Bar Example](image)

This section describes how to create the top-level pull-down menu items. For information about creating the pull-down menu options, see “Creating pull-down menu items” on page 84.

To create a menu bar:

1. Select **Tools > Menus** from the Designer menu to open the Menu Setup dialog box.

2. Type the name of the pull-down menu in the **Caption** entry field. For example, the first menu is usually File; so type File in the **Caption** entry field.

   To create a mnemonic, add a tilde (~) or an ampersand (&) before the character in the **Caption** entry field to which you want to assign the mnemonic.

   Note that top-level menu items cannot have accelerators defined. For more information, see “Menu Item Options” on page 80.

3. *Optional*: Type a value in the **Value** entry field.

   The value can be a symbol (#itemName), a string (itemName), or a number (1, 2, 3, and so on).

4. *Optional*: Type a help ID in the **Help ID** entry field.

   For more information, see “Adding context-sensitive help to an item” on page 95.

5. Click **Add** or **Insert**.

   The item displays in the **Menu Definition** list box.

6. Repeat steps 2-5 for each pull-down menu that you want to create.

7. Save and test the interface.
Menu Item Options

This section describes the selections you can make in the Menu Item Options group box.

Defining an accelerator

You can assign an accelerator to a menu item so you can select the item from the keyboard. Note that you can define a mnemonic for a top-level pull-down menu item, but you cannot define an accelerator for it.

A valid accelerator can contain any combination of the ALT, CTRL, SHIFT, or function keys and an alphanumeric key.

Select an accelerator from the Accelerator drop-down list box, or enter your own accelerator in the Entry Field, as shown in the following figure. For example, the following figure shows how you can define an accelerator (CTRL+O) for the Open menu item:
Defining a mnemonic

You can assign a mnemonic to any menu item so you can select the item from the keyboard. For example, you can open the File menu by pressing ALT+F; then you can select the Open menu option by pressing O.

To define a mnemonic for a top-level pull-down menu option or a menu item, place one of the following characters before the letter for which you want to create a mnemonic:

- A tilde (~)
- An ampersand (&)

To include the tilde character as part of the menu item, type two tilde characters (~). The ampersand does the same thing as a tilde but conforms to Windows usage.

Type a tilde or an ampersand before the appropriate letter in the Caption entry field, as shown here:

Mnemonics also can be used to interact with form items (such as a Button). For more information about creating mnemonics for form items, see “Creating mnemonics for form items” on page 139.
Displaying check marks

You can set up a pull-down menu so that it displays a check mark next to the item that the user selected last. For example, a standard Window menu contains a check mark next to the window that currently has focus.

Cascading menus cannot have check marks.

To add check marks to a pull-down menu:

1. Select the top-level menu item for the pull-down menu for which you want to display check marks. For example, select Edit from the Menu Definition list box.
2. Check the Checked check box in the Menu Item Options group box.
3. Click Replace.

Allowing multiple check marks

You can set up a pull-down menu so that it allows multiple selections (check marks). For example, in a Font Style menu, you might want to allow the user to select Bold and Italic.

To allow multiple selections in a pull-down menu:

1. Select the top-level menu item for the pull-down menu for which you want to allow multiple selections. For example, select Edit from the Menu Definition list box.
2. Check the Multiple Selection check box in the Menu Item Options group box.
   The system checks the Checked check box.
3. Click Replace.

When the user selects an item from a menu that allows multiple selections, the system places a check mark next to each selected item. When the user selects the item again, the system removes the check mark from the item.

Not in Menu Bar menu options

When you check the Not in Menu Bar check box, the menu option does not display in the menu bar. Use this option when you want to create a pop-up menu.

For more information about creating a pop-up menu, see “Creating a pop-up menu” on page 90.
Menu Options

This section describes the selections you can make in the **Menu Options** group box.

Adding a Window menu

The Designer provides a shortcut for adding a standard Window menu to the menu bar. The Window menu displays:

♦ At the far right of the menu bar if there is no **Help** menu
♦ Otherwise, to the immediate left of the **Help** menu

The Window pull-down menu contains the choices **Tile** and **Cascade** and a list of open windows to which the user can set focus.

To add a **Window** menu to the menu bar, check the **Include Window Menu** check box on the Menu Setup dialog box. Note that the menu options do not display in the **Menu Definition** list box, and you cannot edit the default menu options. If you do not want to use the standard **Window** menu, you can create your own.

For information about adding a **Window** menu to an MDI form, see “Adding a Window menu to an MDI form” on page 275.

Adding a Help menu

The Designer provides a shortcut for adding a standard **Help** menu to the menu bar. The **Help** menu displays at the far right of the menu bar. It includes the standard options for a **Help** menu on your operating system.

To include a **Help** menu in the menu bar:

1. Check the **Include Help Menu** check box on the Menu Setup dialog box.
2. In the **Help File** entry field, type the pathname of the Help file. This step links the Help file to the **Help Index** menu option.

Note that the menu options do not display in the **Menu Definition** list box, and you cannot edit the default menu options. If you do not want to use the standard Help menu, you can create your own.
Creating pull-down menu items

After you create a top-level menu (for example File), you can add menu items to it to create the pull-down menus.

To add menu items to a menu:

1. Select **Tools > Menus** from the Designer menu.

   The Menu Setup dialog box displays, as shown in the following figure. Note that the menu options you added previously display in the Menu Definition list box.
2. Select the menu option in the **Menu Definition** list box to which you want to add the menu item. For example, select **File**.

3. Click **Clear** to clear the **Menu Item Options** entry fields.

4. Type the name of the menu item in the **Caption** entry field. For example, to add the **Open menu** item, type Open in the **Caption** entry field.

5. **Optional**: Type a value in the **Value** entry field. The value can be a symbol (#itemName), a string (itemName), or a number.

6. **Optional**: Select an accelerator from the **Accelerator** drop-down list box, or you can enter your own in the Entry Field.

7. **Optional**: Type a help ID in the **Help ID** entry field.

8. Click **Add**. The item displays in the **Menu Definition** list box under **File**.

9. Press the right arrow under the **Menu Definition** list box to demote the **Open** menu item under the **File** menu. Note that a right angle bracket displays before the **Open** menu item in the **Menu Definition** list box. For more information about the **Menu Definition** list box, see “**Menu Definition list box**” on page 86.

10. Repeat steps 2-8 until you have added all menu items to the menu.

11. Save and test the interface.

For information about creating a cascading menu, see “**Creating a cascading menu**” on page 89. For information about creating a pop-up menu, see “**Creating a pop-up menu**” on page 90.
**Adding a separator.** A separator is a horizontal line between menu items. It serves as a visual grouping mechanism to help users more easily read and understand the items on a menu. The user cannot select a separator.

To create a separator, follow the instructions for adding a menu item. In the **Caption** entry field, type a hyphen (−). Click **Add** or **Insert** to add the separator to the **Menu Definition** list box.

**Menu Definition list box**

The **Menu Definition** list box allows you to move items to different levels of a menu hierarchy. For example, consider the **Menu Definition** list shown here:
The **Menu Definition** list box above describes a menu hierarchy with three levels:

1. The items at the top level (**File** and **Edit**) display in the form’s menu bar.
2. The items at the second level (**Open**, **Close**, **Cut**, **Copy**, and **Style**) display as menu items in the **File** and **Edit** menus.
3. The items at the third level (**Font** and **Size**) are a cascading menu under **Style**.

The following figure shows the resulting menu bar:

![Menu Bar Example](image)

**Inserting a menu item**

You can add an item before an existing item. For example, imagine that you have created the items **File**, **Edit**, and **Format**. To insert a **View** item before the **Format** item:

1. Select the **Format** item in the **Menu Definition** list box.
2. Click **Clear**.
3. Click mouse button 1 in the **Caption** entry field and type: **View**
4. Click **Insert**.

The **Menu editor** adds the **View** item before the **Format** item.
**Changing a menu item**

To change a menu item:

1. Select the item that you want to change in the **Menu Definition** list box.
2. Make changes to the Menu Item Options as needed. For example, type a new name for the item in the **Caption** entry field.
3. Click **Replace** to apply the changes.
4. Click **OK** to save your changes or click **Cancel** to exit without making any changes.

Clicking **OK** displays the new name in place of the old one in the menu.

**Removing a menu item**

To remove a menu item:

1. Select the item that you want to remove from the **Menu Definition** list box.
2. Click **Remove**.

The item is removed from the **Menu Definition** list box.

**Promoting and demoting menu items**

To promote an item in the **Menu Definition** list box, press the left arrow under the **Menu Definition** list box. To demote an item in the **Menu Definition** list box, press the right arrow under the **Menu Definition** list box.

To move an item up one position in the **Menu Definition** list box:

1. Select the item.
2. Press the up arrow.

The selected item changes places with the item above it.

To move an item one position down in the **Menu Definition** list box:

1. Select the item.
2. Press the down arrow.

The selected item changes places with the item below it.
Creating a cascading menu

A cascading menu contains a submenu. The following figure shows an example of a cascading menu:

To create the cascading menu shown above:

1. Select Tools > Menus from the Designer menu to display the Menu Setup dialog box.

2. In the Caption entry field, type the name of the cascading menu item. For example, type Style in the Entry Field.
   Fill in the other fields in the Menu Item Options topic box as needed.
   Cascading menus cannot have check marks.

3. Click Add or Insert.

4. Type the name of the menu item you want to add to the cascading menu in the Caption entry field. For example, enter Bold.

5. Click Add or Insert. Make sure that the menu item displays in the right place in the menu hierarchy. If it is not in the right place, use the arrows under the Menu Definition list box to move the menu item where you want it to go.
6. Press the right arrow under the **Menu Definition** list box to demote the menu item under the **Style** menu, as shown here:

![Menu Setup for Form: Test](image)

7. Repeat steps 4-6 for each menu item that you want to add to the cascading menu. For example, add **Italic** and **Underline**.

8. Save and test the interface.

**Creating a pop-up menu**

Creating a pop-up menu involves two stages:

- **Creating the menu.** To create a pop-up menu:
  1. Select **Tools > Menus** from the Designer menu to display the Menu Setup dialog box.
  2. In the **Caption** entry field, type the name of the pop-up menu; for example, DeskTopPopup. The system uses this name, but it is not a name that the user sees.
  3. Check the **Not In Menu Bar** check box.
  4. Fill in other fields in the **Menu Item Options** topic box as needed.
  5. Click **Add** or **Insert**.
  6. For each item in the pop-up menu, type its name in the **Caption** entry field. Repeat steps 3 and 4 for each item you add to the pop-up menu.
6. Press the right arrow under the **Menu Definition** list box to demote each pop-up menu item under the **DeskTopPopup** menu.

For example, in the following figure, the **Open File** and **Close File** menu options are preceded by right-angle brackets, which indicates that these options will display on the **DeskTopPopup** menu:

7. Click **OK** to close the Menu Setup dialog box.
Attaching the menu to a form: To attach a pop-up menu to a form:

1. Set focus to the form where you want to attach a pop-up menu.
2. Select Form > Change form from the Designer menu to display the Change Form dialog box.
3. From the Pop-up Menu drop-down list box, select the name of the pop-up menu to assign to the form.

For example, select DeskTopPopup, as shown here:
4. Click **OK** to save your changes.

5. Save the interface and test it. Press mouse button 2 on the background of the form to display the pop-up menu, as shown here:
Attaching the menu to a form item: To attach a pop-up menu to a form item:

1. Set focus to the form where you wish to attach a pop-up menu to a form item.
2. Select Tools > Methods from the Designer menu.
   The Method Editor appears.
3. Create a postOpenInitialization method for the current form:
   A. Click New on the Method Editor.
   B. Type the following code into the Source edit area, replacing formItem with the name of the form item to which you would like to attach a pop-up menu. Assuming that you are attaching the DeskTopPopup pop-up menu created earlier in this section, the code would be:

   ```small
   postOpenInitialization
   (self controllerItemDict at: #formItem)
   setPopupMenuTo: (self controllerItemDict at: #DeskTopPopup).
   ```
   C. Click Save to save the method.
   D. Click Close to close the Method Editor.
4. Save the interface and test it. Press mouse button 2 on the form item to display the pop-up menu, as shown here:
Adding context-sensitive help to an item

The Designer allows you to add context-sensitive help to an item. You need to create a Help file and use the assigned context strings to link the menu items to the help topics.

The user can get context-sensitive help by:

♦ Selecting the item and pressing F1 to get specific help about the item
♦ Selecting an option on the Help pull-down menu

To add context-sensitive help:

1. Select **Tools > Menus** from the Designer menu to display the Menu Setup dialog box.
2. Select the item where you want to add context-sensitive help.
3. In the **Help ID** entry field, type the context string that corresponds to the help panel to display when the user presses F1.
4. In the **Help File** entry field, type the pathname of the Help file.

The system adds context-sensitive help to the item.

Assigning actions to menu items

Once you create a menu, you must specify the action to take for each item that a user can select. You specify the action by associating a method with each item. The system provides some predefined methods (for example, there is a close method that closes a window).

You also can define your own methods (select **Tools > Methods**). To learn more about creating a method from within the Designer, see “Using the Method Editor to create a method” on page 61.
Assigning a method to a top-level menu item
To assign a method to a top-level pull-down menu:
1. Select **Tools** > **Menus** to display the **Menu Setup** dialog box. The **Menu Definition** list box contains the menu that you defined for the form.
2. Select the menu item where you want to assign a method.
3. Click **Methods**.
   The **Method Assignment** dialog box displays, as shown here:

![Method Assignment dialog box]

4. Select an event type from the **Event Type** radio buttons.
5. Select a method from the **Available Methods** list box and click **Copy**.
   Check the **Inherited Methods** check box if your Designer session is based on a subclass of a subclass of Controller or InterfaceComponent. Checking this Check Box lists all methods defined in the superclass and in the current class.

6. Save and test the interface.
**Triggering methods in response to a menu selection**

You can trigger a different method for each menu item on the menu that the user can select.

To trigger methods in response to a menu selection:

1. Create the menu and the menu items, as described previously. However, you need to enter a valid method name in the Value field for all menu items that you create. These names then are triggered by the menuSelected: method.

   For example, when you create the Open menu item on the File menu, use openFile as the value. Use closeFile as the value for the Close menu item on the File menu.

2. Select **Tools > Methods** to open the Method Editor.

   The Method Editor displays.

3. Create the following method:
   
   A. Click **New** on the Method Editor.
   
   B. Type the following in the Source edit area:
      
      ```text
      menuSelected: menuName
      self perform:
        (cItemDict at: menuName) getValue.
      ```
      
   C. Click **Save** to save the method.
   
   D. Click **Close** to close the Method Editor.

4. Select **File** from the Menu Definition list box.

5. Click **Methods** on the Menu editor to open the Method Assignment dialog box.

6. Select menuSelected: from the **Available Methods** list box to assign the method to the File menu.

7. Click **Copy** to add the method to the Assigned Methods list box.

8. Click **OK** to close the Method Assignment dialog box.

9. Click **OK** to close the Menu Setup dialog box.

10. Save and test the interface.
Toolbar editor

The Designer allows you to create toolbars on a form.

A toolbar provides quick access to frequently performed functions in the application. By default, the system places a toolbar at the top of the form, either under the menu bar, if the form has one, or under the title bar.

Creating a toolbar

To create a new toolbar:

1. Select **Tools > Toolbar** from the Designer menu.

   The Toolbars for Form dialog box displays, as shown here:

2. Click **New**.

   The String dialog box displays, as shown here:

3. Type the name of the toolbar you are creating. You also can accept the default toolbar name.

4. Click **OK**.

   The name of the toolbar displays in the **Toolbars** list box on the Toolbars for Form dialog box.
Editing a toolbar

To edit a toolbar:
1. Select **Tools > Toolbar** from the Designer menu.
2. Select the toolbar that you want to edit from the **Toolbars for Form** list box.
3. Click **Edit**.

The Toolbar Options for Form dialog box displays, as shown here:
The **Name** entry field displays the name of the toolbar. In general, you use the following areas of the Toolbar Options for Form dialog box to work on a toolbar:

- Use the **Item Settings** group box to define individual elements of the toolbar.
- Use the **Toolbar Definition** list box to view the toolbar elements and to arrange items on the toolbar.
- Use the vertical row of Buttons box to move information about individual toolbar elements in the editor.

**Docking:** Docking enables you to create a toolbar that the user can move to any location on the screen. Check the Docking check box to enable this feature.

**Flat:** Flat toolbars have the standard Windows “flat” appearance. Check the Flat check box to enable this feature.

**Item settings**

This section describes the types of toolbar items you can create and provides additional information about settings for toolbar items. These settings are found on the Toolbar Options for Form dialog box.

**Toolbar items.** Items on a toolbar can be any combination of:

- Buttons, containing either text or an image (bitmap). (Clicking a Button initiates an action.)
- Labels, which identify groups of Buttons or provide other information.
- Gaps, which provide visual spacing around other toolbar items.

**Value.** The value allows you to use the getValue method to associate the toolbar with methods to execute when the user clicks a Button on the toolbar. The value can be a symbol (#itemName), a string (itemName), or a number. For more information, see “**Associating the toolbar with methods**” on page 106.

**Tooltips.** The Tooltips entry field allows you to enter text that displays as a tooltip when the user places the cursor over a toolbar Button.

**Check style.** If you select the Check Style check box, the Button on the toolbar displays pressed-in when the user clicks the Button. The Button remains pressed-in until the user clicks the Button again. This change in appearance only works for Image Buttons.
Toolbar Definition list box
The Toolbar Definition list box displays a list of items on the toolbar in the order in which they display. You can use the up arrow and the down arrow located below the List Box to move toolbar items up or down in the hierarchy.

For example, the following figure shows a toolbar that has:

- A label called Press one:
- A gap of 10 units
- A Print button
- A gap of 10 units
- A system Button called Info

You can use the arrow Buttons that display below the Toolbar Definition list box to rearrange items on the toolbar.
Adding items to the Toolbar Definition list box. To add items to the Toolbar Definition list box, click one of the vertical Buttons as described in the following table:

<table>
<thead>
<tr>
<th>Click</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Add the item in the Item Settings group box to the end of the Toolbar Definition list box.</td>
</tr>
<tr>
<td>Insert</td>
<td>Insert the item in the Item Settings group box before the item in the Toolbar Definition list box that is selected.</td>
</tr>
<tr>
<td>Replace</td>
<td>Change the item in the Toolbar Definition list box that is selected, assigning it the attributes in the Item Settings group box.</td>
</tr>
<tr>
<td>Remove</td>
<td>Remove the item in the Toolbar Definition list box that is selected.</td>
</tr>
<tr>
<td>Clear</td>
<td>Clear the Item Settings group box.</td>
</tr>
</tbody>
</table>
Resizing the toolbar

If the toolbar is too short to accommodate the bitmap Button, click Options (described in the following section) to resize the toolbar. When you add an icon to the toolbar, the image is not resized automatically (the default image size is 16x16). To resize icons automatically, click Options and check the Resize Icon Based on Toolbar Image Size check box.

Toolbar options

Click Options to display the Form Toolbar Misc Options dialog box. The following table describes the options on the dialog box:

<table>
<thead>
<tr>
<th>Select option</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom check box</td>
<td>Place the toolbar at the bottom of the form. The default is to place the toolbar at the top of the form under the menu bar (if there is a menu bar).</td>
</tr>
<tr>
<td>Hidden During OLE In-place Activation</td>
<td>Hide the toolbar when an OLE in-place activation (visual editing) occurs. The toolbar is replaced with the toolbar of the linked application. For more information about visual editing, refer to the ObjectStudio OLE User's Guide, P40-3805.</td>
</tr>
<tr>
<td>Resize Icon Based on Toolbar Image Size check box</td>
<td>Resize an icon on a Button to fit the toolbar dimensions automatically. You also can change the dimensions in the Dimensions entry fields.</td>
</tr>
<tr>
<td>Dimensions entry fields</td>
<td>Define the height and width of the toolbar, an image, or a Button (in pixels).</td>
</tr>
</tbody>
</table>
Creating a toolbar example

To create the toolbar shown here:

1. Select **Tools > Toolbar** to create a new toolbar as described in “Creating a toolbar” on page 98.

2. Click **Edit** on the Toolbars for Form dialog box.
   The Toolbar Options for Form dialog box displays.

3. Create a label:
   A. Select the **Label** radio button in the **Item Settings** topic box.
   B. Type the following label in the **Label** entry field:
      `Press one:`
   C. Click **Add** or **Insert**.
   The label displays in the **Toolbar Definition** list box.

4. Create a gap:
   A. Select the **Gap** radio button in the **Item Settings** topic box.
   B. Type a gap specification in the **Width** entry field. For example, enter `10` in the **Width** entry field.
   C. Click **Add** or **Insert**.
   The gap specification displays in the **Toolbar Definition** list box.
5. Create a Button with text:
   A. Select the Button radio button in the Item Settings topic box.
   B. Type a return value in the Value entry field. Note that you can leave this field blank for now.
   C. In the Label entry field, type:
      Print
   D. Type a brief hint in the Tooltip entry field that describes the function of the Button; for example:
      Print file
   E. Click Add or Insert.
   The Button is added to the Toolbar Definition list box.

6. Create another gap, as described in step 4.

7. Create a Button with an image:
   A. Select the Image radio button in the Item Settings list box.
   B. Type a return value in the Value entry field. Note that you can leave this field blank for now.
   C. Select an image type from the Image Type radio buttons. For example, select the System Icon radio button.
   D. Select the icon type from the Spec pull-down list box (for example, select Info).
      The icon displays in the Example topic box.
   E. Resize the image to fit the button. To do so, click Options. The Form Toolbar Misc Options dialog opens. On the Form Toolbar Misc Options dialog, check the Resize Icon Based On Toolbar Image Size checkbox. Click OK.
   F. On the Toolbar Options dialog, click Add or Insert.

When you define a toolbar element that has an image on a Button, check the Check Style check box to give the Button a pressed-in appearance when the Button is selected.

8. Click OK.
   The form displays with the toolbar that you just created.
Associating the toolbar with methods

The following procedure provides an example of how to associate the toolbar with a method that executes when the user clicks a Button:

1. Select Tools > Methods to open the Method Editor.

   Use the Method Editor to create a method that sends getValue to the toolbar control item to determine which Button was clicked. Provide logic in the method to handle each Button click.

2. Create the following method:

   A. Click New on the Method Editor.
   B. Type a method in the Source edit area. For example, the following method displays text on the status line when a Button on the toolbar is clicked:

      ```
      handleToolbarBtns
      | val |
      val := formToolbar1 getValue.
      (val = #button1) ifTrue:
      [ formStatusLine1 setPaneText:'Prints the file' at:1. ]
      ...
      ```
   
   C. Click Save to save the method.
   D. Click Close to close the Method Editor.

3. Select Tools > Toolbar to open the Toolbars for Form dialog box. Select the toolbar where you want to attach the method and click Edit to open the Toolbar Options for Form dialog box.

4. Click Methods to open the Method Assignment dialog box.

5. Select a method from the Available Methods list box and copy it to the Assigned Methods list box.

   For example, select the handleToolbarBtns method from the Available Methods list box. Click Copy to add the method to the Assigned Methods list box.

6. Click OK to close the Method Assignment dialog box.

7. Save and test the interface.

   Note that the method you create handles the entire toolbar (not each toolbar Button individually). For more information about the Method Editor, see "Using the Method Editor to create a method" on page 61.
Status line editor

The Designer allows you to add one or more status lines to a form. A status line provides a mechanism for displaying status information and user hints on a form.

By default, a status line is placed at the bottom of the form. However, you can place the status line at the top of the form.

Status line segments: The information in a status line can be displayed in one or several segments. The only information you can display on the status line is text.

Creating a status line

To create a status line:

1. Select Tools > Status Line from the Designer menu.

The Status Lines for Form dialog box displays, as shown here:
2. Click **New**.

The String dialog box displays, as shown here:

![String Dialog](image)

3. Type the name of the status line that you are creating. You can accept the default status line name.

4. Click **OK**.

The system places the name of the status line in the **Status Lines** list box.
Editing a status line

To edit an existing status line:

1. Select Tools > Status Line from the Designer menu.
2. Select the status line that you want to edit from the Status Lines list box.
3. Click Edit.

The Status Line Options for Form dialog box displays, as shown here:
The name of the status line is displayed in the **Name** entry field. In general, you use the following areas of the editor to work on a status line:

- Use the **Section Settings** group box to define individual elements of the status line.
- Use the vertical row of buttons to move information about individual status line elements in the editor.
- Use the **Wd, Indicators**, and **Styles** list boxes to view the Status Line items and to rearrange the order.

**Note:**

- When you create a new status line, a default text area is already defined. You can remove or edit the text area.
- Check the **Top** check box to display the status line at the top of the form.
- You can create more than one status line for a form.
- The Stretch style allows one indicator (segment) on the status line to expand to fill up the unallocated space on the status line. Note that the width entered in the Width entry field is ignored. You can assign the Stretch style to only one indicator on a status line.
- Use the **Display Effect** check boxes to set the display of each segment. Segments of the status line can display flat, sunken, or raised.
- Use the up arrow and the down arrow to rearrange indicators in the hierarchy. You also can select and drag an item to the desired position.
Creating a status line example
To create the status line in the window shown here:

1. Select Tools > Status Line to create a new status line as described in “Creating a status line” on page 107.
2. Click Edit on the Status Lines for Form dialog box to open the Status Line Options for Form dialog box.
3. Change the style for the Text item:
   A. Check the Stretch check box in the Style topic box.
   B. Click Replace.
4. Select the Caps Lock radio button in the Indicator topic box.
5. Check the Sunken check box in the Display Effect topic box.
6. Type a different width in the Width entry field or leave the default width of 100 pixels.
7. Click Add or Insert.
8. Repeat steps 4-7 to add the Num Lock and Scroll Lock indicators to the status line.
9. Click OK to close the Status Line Options for Form dialog box.

The status line displays on the form. Note that you do not need to be in Test Interface mode to test the status line.
Displaying text in the status line

You can display a string in the text area of the status line. For example, when the user clicks a Button, you can display text in the status line that explains what the Button is doing.

To set a text string in a status line:

1. Select FormItem > New Item to open the New Item dialog box.
2. Add a Button to the form with the label Print.
3. Select Tools > Methods from the Designer menu.
   The Method Editor displays.
4. Create the following method:
   A. Click New on the Method Editor.
   B. In the Source edit area type:

   ```
   addText
   formStatusLine1 setPaneText:'Print the file' at:1.
   ```
   C. Click Save to save the method.
   D. Click Close to close the Method Editor.
5. Double-click the Print button.
   The Form Button Options dialog box displays.
6. Click Methods to open the Method Assignment dialog box.
7. Select the addText method from the Available Methods list box.
8. Click Copy to add the method to the Assigned Methods list box.
9. Click OK to close the Method Assignment dialog box.
10. Click OK to close the Form Button Options dialog box.
11. Save and test the interface.

When you click Print, the text ‘Print the file’ displays on the status line.
Tooltip editor

This section describes tooltips and how to add tooltips to form items. For more information about adding tooltips to Buttons on a toolbar, see “Tooltip editor” on page 98.

What are Tooltips?

Tooltips provide hints about the function of a Button on a toolbar or a form item. A tooltip displays when you place the cursor over the Button or form item and keep it there for a few seconds. At the same time, text can display on the status line that further explains the function of the item.

Creating tooltips for form items

Before you can create tooltips for form items, you first must add the form items to the form. For more information about adding form items to a form, see “Creating a form item” on page 125.

To create a tooltip:

1. Select Tools > Tooltip from the Designer menu.

The Tooltip Options for Form dialog box displays, as shown here:
2. Select the controller item from the Controller Items list box where you want to add a tooltip.

3. Type the text that you want to display on the tooltip in the Tooltip String entry field. The tooltip string also displays simultaneously in the Tooltip Strings list box.

   Click Delete Tooltip if you decide that you do not want to add a tooltip or make a change to the existing tooltip.

4. Repeat steps 2 and 3 for each controller item to which you want to add a tooltip.

5. Click OK to close the Tooltip Options for Form dialog box.

6. Save and test the interface.

Creating property pages and property sheets

The Designer allows you to create property pages (forms) and a container for the pages called a property sheet. Property sheets are like Notebooks and contain a set of tabbed pages that usually have related functionality. The following figure shows an example of a property sheet:

![Property Sheet Example]

The following sections describe how to create property pages and a property sheet.

---

User Interface Guide, P40-3811
Chapter: 2. Creating forms
Section: Creating property pages and property sheets

114
Creating a property page

To create a property page:

1. Select Form > New Property Page from the Designer menu.

   The String dialog box displays, as shown here:

   ![String Dialog](image)

2. Type a name for the new property page. For example, enter Background in the Enter new property page name entry field.

3. Click OK.

   A blank property page form displays.

4. Repeat steps 1–3 for each property page that you want to create. For this example, create at least one more property page called Settings.

   The logical limit for the number of property pages on a property sheet in ObjectStudio is 50. However, there is a limit to the number of tabs that will fit on the screen.

After you create the property page forms, you can use the New Item dialog box to add items to each form. For more information about adding form items, see “Creating a form item” on page 125.
Creating a property sheet

To create a property sheet:

1. Select Form > Property Sheet from the Designer menu.

   The Property Sheets dialog box displays, as shown here:

   ![Property Sheets dialog box]

2. Click New.

   The String dialog box displays, as shown here:

   ![String dialog box]

3. Type the name of the property sheet in the Enter Property Sheet name entry field. For example, type Properties.

4. Click OK.
Adding property pages to a property sheet

To add pages to a property sheet or to edit a property sheet:

1. Select Form > Property Sheet from the Designer menu to display the Property Sheets dialog box.
2. From the Sheets list box, select the property sheet that you want to edit. For example, select Properties.
3. Click Edit.

The Property Sheet Options dialog box displays, as shown here:

4. Select the property page that you want to add to the property sheet from the Available Pages list box. For example, select the Background property page.
5. Click Add or Insert.

The property page displays in the Page Definition list box.
6. Repeat steps 4 and 5 for each property page that you want to add to the property sheet. For this example, select the Settings property page.

7. Click OK to close the Property Sheet Options dialog box.

You can rearrange the order of the property pages that display in the Page Definition list box. Use the up and down arrows that display below the List Box to promote or demote the selected page.

The Modal check box is checked by default. If you do not want the property sheet to be a modal dialog box, uncheck the Modal check box. For more information about modality, see “Modal dialog boxes” on page 59.

Displaying the property sheet

Once you have created the property sheet, you must provide a way for the user to access the property sheet. You can create a Button or a menu item that displays the property sheet.

Using a Button to display a property sheet

To use a Button to open a property sheet:

1. Create a Button on the main form. For example, create a Button and label it Properties.

2. Double-click the Button to open the Form Button Options dialog box.

3. Select Open Form from the Custom drop-down list box.

4. Click Search to open the Selection Form dialog box.

5. Select the name of the form to which you want to link the property sheet. For example, select Properties.

6. Click Set.

   Note that Properties now displays in the Object Name entry field.

7. Click OK to close the Form Button Options dialog box.

8. Save and test the interface.

The property sheet does not display until you test the interface.
Using a menu option to display a property sheet

To display a property sheet from a menu option:

1. Use the Menu Setup dialog box to create a pull-down or pop-up menu option to open the property sheet. For example, create a File pull-down menu with an option named Properties.
2. Select Tools > Methods to open the Method Editor.
3. Create the following method:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:

   ```plaintext
   menuSelected: menuName
   self perform: (cItemDict at: menuName) getValue.
   ```
   C. Click Save to save the method.
4. Create the following method:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:

   ```plaintext
   openPropertySheet
   (formDict at: #Properties) open.
   ```
   C. Click Save to save the method.
   D. Click Close to close the Method Editor.
5. Select Tools > Menus to open the Menu Setup dialog box.
6. Select File from the Menu Definition list box.
7. Click Methods to open the Method Assignment dialog box.
8. Select menuSelected from the Available Methods list box.
9. Click Copy to add the method to the Assigned Methods dialog box.
10. Click OK to close the Method Assignment dialog box.
11. Click OK to close the Menu Setup dialog box.
12. Save and test the interface.

The property sheet called Properties displays when you select Properties from the File pull-down menu.
3. Creating form items

Form item overview

A form item is a visual item that you can place on a form. The Designer’s New Item dialog box contains a Value Set from which you select form items to place on a form.

This section provides an overview of form items and information about their Smalltalk implementation. For an overview about interfaces, controllers, controller items, and interface components, see “Designer concepts” on page 13.
The following table gives an overview of the form items that the Designer provides. For detailed information about each form item, see “Form item descriptions” on page 163.

<table>
<thead>
<tr>
<th>Form item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap Selector</td>
<td>Interface component that allows the user to select a bitmap.</td>
</tr>
<tr>
<td>Button</td>
<td>Allows the user to initiate an action; for example, to open or close a form.</td>
</tr>
<tr>
<td>Chart</td>
<td>Displays numeric information in a Bar Chart, a Line Graph, or a Pie Chart.</td>
</tr>
<tr>
<td>Check Box</td>
<td>Mechanism for toggling a choice on or off.</td>
</tr>
<tr>
<td>Check List</td>
<td>Group of Check Boxes representing choices that are not mutually exclusive.</td>
</tr>
<tr>
<td>Combo List</td>
<td>Combination of an Entry Field and a List Box; the user can select from the List Box or type a choice that is not in the list.</td>
</tr>
<tr>
<td>Date Field</td>
<td>Entry Field that only displays and accepts text in a date format.</td>
</tr>
<tr>
<td>Draw Pad</td>
<td>Displays bitmap graphics; provides an area for the user to draw graphical images.</td>
</tr>
<tr>
<td>Drop-down Combo Box</td>
<td>Combination of an Entry Field and a Drop-down List Box; the user can select from the List Box or type a choice that is not in the list.</td>
</tr>
<tr>
<td>Drop-down List Box</td>
<td>Space-saving List Box with two states: (1) display only the current selection or (2) display the list.</td>
</tr>
<tr>
<td>Entry Field</td>
<td>One-line field that displays text and accepts user input.</td>
</tr>
<tr>
<td>Horizontal Border</td>
<td>Horizontal moveable border that enables a user to resize lists, Tree Views, text fields, or other items.</td>
</tr>
<tr>
<td>Line</td>
<td>Straight or rounded line that separates a form into sections.</td>
</tr>
<tr>
<td>List Box</td>
<td>Scrollable list from which the user can select one or more items.</td>
</tr>
<tr>
<td>ListView</td>
<td>Box that displays a list divided into resizable, labeled columns. The user can select one of the items and press ENTER or can double-click on it.</td>
</tr>
<tr>
<td>Multi Line Entry Field</td>
<td>Field that displays text and accepts user input. It can have an unlimited number of lines.</td>
</tr>
<tr>
<td>Notebook</td>
<td>Collection of logically related pages where you can place form items.</td>
</tr>
<tr>
<td>Number Field</td>
<td>Entry Field that only displays and accepts text in a numeric format.</td>
</tr>
<tr>
<td>OLE Custom Control</td>
<td>Mechanism for inserting an OLE control.</td>
</tr>
<tr>
<td><strong>Form item</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>OLE Item</td>
<td>Provides a site for an embedded or linked OLE object.</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>Indicates how much progress has been made for a particular task.</td>
</tr>
<tr>
<td>Radio Buttons</td>
<td>Mechanism for toggling a set of mutually exclusive choices.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Displays related information in a two-dimensional display.</td>
</tr>
<tr>
<td>Scatter Graph</td>
<td>Displays sets of related number pairs on a graph.</td>
</tr>
<tr>
<td>Slider</td>
<td>Graphical display that allows the user to set a discrete value within a range.</td>
</tr>
<tr>
<td>Spin Button</td>
<td>Allows the user to select an item from a closed set of information (for example, months of the year).</td>
</tr>
<tr>
<td>Static Text</td>
<td>Text that labels information on a form.</td>
</tr>
<tr>
<td>Tabular List Box</td>
<td>List Box that presents items as columns of data.</td>
</tr>
<tr>
<td>Text Collector</td>
<td>Scrollable box that contains text.</td>
</tr>
<tr>
<td>Text Editor</td>
<td>Extensible editor that includes regular expression searches and text formatting options.</td>
</tr>
<tr>
<td>Topic Box</td>
<td>Labeled outline box that contains a related set of screen elements.</td>
</tr>
<tr>
<td>Tree View</td>
<td>Hierarchical display of items; the user can expand and contract levels in the hierarchy.</td>
</tr>
<tr>
<td>Value Set</td>
<td>Graphical array of items from which the user can select or drag one item at a time.</td>
</tr>
<tr>
<td>Vertical Border</td>
<td>Vertical moveable border that enables a user to resize lists, Tree Views, text fields, or other items.</td>
</tr>
<tr>
<td>Workplace List Handler</td>
<td>Used for creating workplace applications; acts as a container for other objects.</td>
</tr>
<tr>
<td>Workplace Object Handler</td>
<td>Used for creating workplace applications; object that stores data or performs an action.</td>
</tr>
<tr>
<td>Workplace Object Template</td>
<td>Used for creating workplace applications; a template for creating new workplace objects.</td>
</tr>
</tbody>
</table>

**Smalltalk:** Smalltalk implements each form item in two parts:

- An instance of a subclass of FormItem
- An instance of a subclass of ControllerItem
For example, if you place a Button on a form, the system implements it by creating an instance of the classes:

♦ FormButton (a subclass of FormItem)
♦ ButtonCtrl (a subclass of ControllerItem)

The following table shows the correspondence between items on the Value Set, their FormItem class, and their ControllerItem class:

<table>
<thead>
<tr>
<th>Value Set name</th>
<th>FormItem class</th>
<th>ControllerItem class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap Selector</td>
<td>ICBitmapSelector</td>
<td>None</td>
</tr>
<tr>
<td>Button</td>
<td>FormButton</td>
<td>ButtonCtrl</td>
</tr>
<tr>
<td>Chart</td>
<td>FormBarChart</td>
<td>BarChartCtrl</td>
</tr>
<tr>
<td>Check Box</td>
<td>FormCheckBox</td>
<td>CheckBoxCtrl</td>
</tr>
<tr>
<td>Check List</td>
<td>FormCheckList</td>
<td>MSListCtrl</td>
</tr>
<tr>
<td>Combo List</td>
<td>FormCombo</td>
<td>ComboCtrl</td>
</tr>
<tr>
<td>Date Field</td>
<td>FormDate</td>
<td>DateCtrl</td>
</tr>
<tr>
<td>Draw Pad</td>
<td>FormDrawPad</td>
<td>DrawPadCtrl</td>
</tr>
<tr>
<td>Drop-down Combo Box</td>
<td>FormDropCombo</td>
<td>ComboCtrl</td>
</tr>
<tr>
<td>Drop-down List Box</td>
<td>FormDropList</td>
<td>DropDownListCtrl</td>
</tr>
<tr>
<td>Entry Field</td>
<td>FormString</td>
<td>StringCtrl</td>
</tr>
<tr>
<td>Horizontal Border</td>
<td>FormHorizontalBorder</td>
<td>MoveableBorderCtrl</td>
</tr>
<tr>
<td>Line</td>
<td>FormLine</td>
<td>LineCtrl</td>
</tr>
<tr>
<td>List Box</td>
<td>FormList</td>
<td>ListCtrl</td>
</tr>
<tr>
<td>ListView</td>
<td>FormListView</td>
<td>ListViewCtrl</td>
</tr>
<tr>
<td>Multi Line Entry Field</td>
<td>FormMlEntryField</td>
<td>MlEntryFieldCtrl</td>
</tr>
<tr>
<td>Notebook</td>
<td>FormNoteBook</td>
<td>NoteBookCtrl</td>
</tr>
<tr>
<td>Number Field</td>
<td>FormNumber</td>
<td>NumberCtrl</td>
</tr>
<tr>
<td>Value Set name</td>
<td>FormItem class</td>
<td>ControllerItem class</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>OLE Custom Control</td>
<td>FormOleControl</td>
<td>ControlCtrl</td>
</tr>
<tr>
<td>OLE Item</td>
<td>FormEmbed</td>
<td>EmbedCtrl</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>FormProgressBar</td>
<td>ProgressBarCtrl</td>
</tr>
<tr>
<td>Radio Buttons</td>
<td>FormRadioButton</td>
<td>RadioButtonCtrl</td>
</tr>
<tr>
<td>Rectangle</td>
<td>FormRect</td>
<td>RectCtrl</td>
</tr>
<tr>
<td>Scatter Graph</td>
<td>FormScatterGraph</td>
<td>TwoDGraphCtrl</td>
</tr>
<tr>
<td>Slider</td>
<td>FormSlider</td>
<td>SliderCtrl</td>
</tr>
<tr>
<td>Spin Button</td>
<td>FormSpinButton</td>
<td>SpinButtonCtrl</td>
</tr>
<tr>
<td>Static Text</td>
<td>FormStatic</td>
<td>ControllerItem</td>
</tr>
<tr>
<td>Tabular List Box</td>
<td>FormTabList</td>
<td>TabListCtrl</td>
</tr>
<tr>
<td>Text Collector</td>
<td>FormTextCollector</td>
<td>TextCollectorCtrl</td>
</tr>
<tr>
<td>Text Editor</td>
<td>FormTextEditor</td>
<td>TextEditorCtrl</td>
</tr>
<tr>
<td>Topic Box</td>
<td>FormTopicBox</td>
<td>TopicBoxCtrl</td>
</tr>
<tr>
<td>Tree View</td>
<td>FormTreeView</td>
<td>TreeViewCtrl</td>
</tr>
<tr>
<td>Value Set</td>
<td>FormValueSet</td>
<td>ValueSetCtrl</td>
</tr>
<tr>
<td>Vertical Border</td>
<td>FormVerticalBorder</td>
<td>MoveableBorderCtrl</td>
</tr>
<tr>
<td>Workplace List Handler</td>
<td>FormListHandler</td>
<td>ListHandlerCtrl</td>
</tr>
<tr>
<td>Workplace Object Handler</td>
<td>FormObjectHandler</td>
<td>ObjectHandlerCtrl</td>
</tr>
<tr>
<td>Workplace Object Template</td>
<td>FormWorkplaceTemplate</td>
<td>WorkplaceTemplateCtrl</td>
</tr>
</tbody>
</table>
Working with form items

This section describes the basic operations that you can perform with form items. This section does not describe individual form items, nor does it discuss how to customize form items. To learn more about these topics, see “Form item descriptions” on page 163.

Creating a form item

The Designer allows you to use different ways to create a form item. Develop your own style according to personal preference or the type of form item you are creating. This section describes different ways to create a form item.

To create a form item:

1. Set focus to the form to which you want to add form items.
2. Select FormItem > New Item from the Designer menu.

The New Item dialog box displays, as shown here:
The New Item dialog box contains a Value Set, or pictorial array, of form items that you can add to a form. They are arranged alphabetically. When you select a form item, a box displays around the item, and the name of the item is displayed under the Name entry field.

The Name entry field allows you to name a form item before you create it. When you type text in this field, the system assigns the text to:

- The instance variable, if one is created, that is associated with the form item
- Any text that the form item displays
- A corresponding item in the controller’s item dictionary, cItemDict

If you do not supply the name at creation, you can always add it later. You also can edit it later. You also can make the display text different from the name.

Following are some different ways to create a form item:

- **Alternative 1, Double-click the icon.** To create a form item:
  1. *Optional:* Type text in the **Name** entry field.
  2. Double-click the appropriate icon in the New Item dialog box.
  The new form item displays on the form.

- **Alternative 2, Drag the icon to the form.** To create a form item:
  1. *Optional:* Type text in the **Name** entry field.
  2. Select the icon of the form item that you want to create.
  3. Press mouse button 1, and drag the form item to the form.
  The new form item displays on the form.
Alternative 3, Click OK. To create a form item:

1. Optional: Type text in the Name entry field.
2. Select the icon for the form item that you want to create.
3. Click OK.

The new form item displays on the form.

Alternative 4, Create by copying. The Designer allows you to create a form item by copying another form item. All characteristics of the form item, except its name and location on the form, are copied when you copy a form item. To copy a form item:

1. Select the form item that you want to copy.

You can select multiple items to copy; to learn how, see “Selecting a form item” on page 129.

2. Select FormItem > Copy Item(s) from the Designer menu.
   - If you selected just one form item, go to step 3.
   - If you selected more than one form item, the Designer makes a copy of each form item and automatically renames the form items.

3. The String dialog box prompts you to supply a name for the new item (all form items on an interface must have a unique name). Type the new name in the Enter name for new item entry field.
4. Click OK to close the String dialog box.

The new form items display on the form.
Creating instance variables

The New Item dialog box contains an **Instance Variable** check box. When it is checked, the system creates an instance variable for the form item when it generates code for the interface. When you write Smalltalk code for the interface, you can use the instance variable to refer to the controller item.

By default, the system checks or unchecks the **Instance Variable** check box depending on the type of form item you are creating. For example, because you rarely need access to a Static Text item or to a Line, the system does not create instance variables for these form items. The Designer allows you to override the default setting for all form items.

If you do not create an instance variable for a form item, you can create one later by using the Instance Variables dialog box. For more information, see “Working with instance variables” on page 270.

**Naming an instance variable**

By default, the system provides a name for each instance variable. Alternately, you can choose to supply an instance variable name when you create a form item.

To change the instance variable name:

1. Double-click the form item.
   - The Form Options dialog box displays.
2. Type the new name in the **Name** entry field.

   **The Smalltalk convention is that instance variable names start with a lowercase letter.**
Selecting a form item

The following table describes how to select form items:

<table>
<thead>
<tr>
<th>To</th>
<th>Do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select one form item</td>
<td>Click the item using mouse button 1.</td>
</tr>
<tr>
<td>Deselect a form item</td>
<td>Select FormItem &gt; Deselect Item(s) from the Designer menu.</td>
</tr>
<tr>
<td>Select all form items on a form</td>
<td>Select FormItem &gt; Select all from the Designer menu.</td>
</tr>
<tr>
<td>Select some of the form items on a form. (Alternative 1)</td>
<td>Select one item; then press and hold the CTRL key as you select subsequent items.</td>
</tr>
<tr>
<td>Select some of the form items on a form. (Alternative 2)</td>
<td>Press mouse button 1 on the background of the form and drag the mouse. A rectangle displays as you drag. When you release the mouse button, the items that are enclosed completely by the rectangle are selected.</td>
</tr>
</tbody>
</table>

When you select one item, the system highlights it. When you select multiple items, the system highlights all of them but assigns a special highlight to the last item you select.

**Selection head:** The item that receives special highlighting is the selection head. This item receives special treatment when you align or resize groups of form items. For more information about aligning form items, see “Aligning form items” on page 133.

Deleting a form item

To delete a form item:

1. Select the item or items that you want to delete.
2. Select FormItem > Delete Item(s) from the Designer menu.

The system asks you to confirm whether you really want to delete the item or items. Click OK to delete the items or Cancel to stop the deletion.
Using the grid

The Designer provides a grid that you can use when you want to place objects more accurately on a form.

To use the grid, select Form > Grid and snap from the Designer menu. The Grid and Snap dialog box displays, as shown here:

Options

The following table describes the options that you can set on the Grid and Snap dialog box:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid Resolution</td>
<td>Specifies, in tenths of millimeters, the distance between grid lines. If you decrease the number, the system places the grid lines closer together, allowing you to place items with more precision. If you increase the number, the system places the grid lines farther apart.</td>
</tr>
<tr>
<td>Show grid</td>
<td>Displays the grid on the form when checked. The grid is a collection of Horizontal and Vertical Lines.</td>
</tr>
<tr>
<td>Snap to grid</td>
<td>Restricts the placement of objects when checked. The top and left edges of the object are on grid lines.</td>
</tr>
</tbody>
</table>
Exiting the grid and Snap dialog box

Click **OK** to exit the dialog box and save your changes. Click **Cancel** to exit without making any changes.

**Overriding Snap to grid**

If you check Snap to grid, you can override it:

1. Select the item or items that you want to move or resize.
2. Hold down the **CTRL** key while you drag the selection with mouse button 1.

**Resizing a form item**

If you check Snap to grid on the Snap and Grid dialog box and you resize a form item, all sides of the form item align to a grid line.

To resize a form item:

1. Select the item.
2. Do one of the following:
   A. Use mouse button 1 to drag one of the item’s selection handles until the item is the size you want.
   B. Press Page Up to shrink the item and press Page Down to enlarge the item by one grid size each time you press Page Up or Page Down.
Placing a form item

This section shows you how to move form items.

Moving a form item with the mouse

To use the mouse to move a form item:

1. Select the item or items that you want to move.
2. Use mouse button 1 to drag the selection to its new location.

Moving a form item with the keyboard

To use the keyboard to move an item by pixels:

1. Select the item that you want to move.
2. Hold down the SHIFT key while you press an arrow key.
   The item moves one pixel in the appropriate direction.

To use the keyboard to move an item to the grid:

1. Select Form > Grid and Snap to open the Grid and Snap dialog box.
2. Check the Show grid and Snap to grid check boxes.
3. Click OK to close the Grid and Snap dialog box.
4. Select the item that you want to move.
5. Hold down the ALT key while you press the appropriate arrow key (up, down, left, or right).

   The item moves one grid at a time in the direction of the arrow key that you press. The left and top sides of the item align to the next grid line in the respective direction.
Aligning form items

The Designer contains the Alignment toolbar, which helps you arrange and size groups of form items. The Alignment toolbar displays just under the Designer menu. You also can select FormItem > Align to display a submenu that contains the same functions as the Alignment toolbar.

The toolbar is divided into three parts, as described in the following table:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Allows you to</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Align form items vertically.</td>
</tr>
<tr>
<td>H</td>
<td>Align form items horizontally.</td>
</tr>
<tr>
<td>S</td>
<td>Resize form items.</td>
</tr>
</tbody>
</table>

Also note that:

♦ Tooltips and the status line provide hints about each Button on the Alignment toolbar. To display a hint about a Button on the toolbar, place the cursor on the Button about which you want information. Wait for one second, and the system displays a tooltip. The status line also can display information about the Button.

♦ The Alignment toolbar is designed to work with multiple form items that you select. If you need a reminder about how to select multiple form items, see “Selecting a form item” on page 129.

♦ When you select multiple form items, the system highlights the last item you select differently from the rest. The different form item is the selection head, and it is used in some of the alignment and resizing operations.
### Aligning vertically

The following table describes each vertical alignment tool on the Alignment toolbar:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>For all selected form items</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Icon]</td>
<td>Align items top justified</td>
<td>Aligns the form item tops to the top of the selection head.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Align items most top justified</td>
<td>Aligns the form item tops to one grid line below the top of the form.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Align items bottom justified</td>
<td>Aligns the form item bottoms to the bottom of the selection head.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Align items most bottom justified</td>
<td>Aligns the form item bottoms to one grid line above the bottom of the form.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Center items vertically</td>
<td>Aligns the form item centers to the imaginary horizontal line that divides the form into vertical halves.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Tile items vertically</td>
<td>Inserts an equal amount of vertical space between each item.</td>
</tr>
</tbody>
</table>
### Aligning horizontally

The following table describes each horizontal alignment tool on the Alignment toolbar:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>For all selected form items</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Icon]</td>
<td>Align items left justified</td>
<td>Aligns the left sides of the form items to the left side of the selection head.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Align items most left justified</td>
<td>Aligns the left sides of the form items to one grid line away from the left side of the form.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Align items right justified</td>
<td>Aligns the right sides of the form items to the right side of the selection head.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Align items most right justified</td>
<td>Aligns the right sides of the form items to one grid line away from the right side of the form.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Center items horizontally</td>
<td>Aligns the centers of the form items to the imaginary vertical line that divides the form into horizontal halves.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Tile items horizontally</td>
<td>Inserts an equal amount of horizontal space between each item.</td>
</tr>
</tbody>
</table>

### Resizing

This table describes resizing tools on the Alignment toolbar:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>Resizes all selected form items</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Icon]</td>
<td>Resize horizontally to same size</td>
<td>To the width of the selection head.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Resize vertically to same size</td>
<td>To the height of the selection head.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Resize to same size</td>
<td>To the width and height of the selection head.</td>
</tr>
</tbody>
</table>
Changing the class of a form item

You can change a form item from one class to another. For example, you can change a List Box to a Drop-down List Box.

Of course, if you change a form item to a new class that has very little in common with the original class, you can lose some information about the original form item. For example, if you change a List Box to a Button, the contents of the List Box are lost.

To change the class of an item:

1. Select the form item whose class you want to change.

2. Select FormItem > Change Class from the Designer menu.

   The Basic Item Properties dialog box displays, as shown here:

   3. Select the new class using the Class spin button.

   4. If necessary, adjust the size and placement of the form item in the Geometry in 1/10 mm topic box.

   5. Click OK to exit and save your changes, or click Cancel to exit without making any changes.
Customizing form items

This section describes how to customize form items within the Designer.

Do one of the following to customize a form item:

♦ Double-click the form item.
♦ Select the form item and press ENTER.
♦ Select the form item and select FormItem > Change Options from the Designer menu.

When you perform one of the previous actions, an options dialog box specific to the form item opens. The options dialog boxes that are specific to each form item are described in the online Help.

Most options dialog boxes have Buttons as described in the following table:

<table>
<thead>
<tr>
<th>Button</th>
<th>Press it to display</th>
<th>To change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>The Display Format dialog box.</td>
<td>The display of the form item (for example, font and color).</td>
</tr>
<tr>
<td>Link</td>
<td>The Link Hookups dialog box.</td>
<td>The data source for a form item.</td>
</tr>
<tr>
<td>Update</td>
<td>The Link Hookups dialog box.</td>
<td>The data source or destination for a form item whose value the user can change.</td>
</tr>
<tr>
<td>Link</td>
<td>The Method Assignment dialog box.</td>
<td>The methods to execute when a form item event occurs.</td>
</tr>
<tr>
<td>Methods</td>
<td>The Form Item Options dialog box.</td>
<td>Form item attributes and to which keys the form item responds.</td>
</tr>
<tr>
<td>Options</td>
<td>The Form Item Options dialog box.</td>
<td></td>
</tr>
</tbody>
</table>
Customizing multiple form items

To customize the display of a set of form items:

1. Select multiple form items on a form.
2. Select FormItem > Change Options from the Designer menu.

The Options Multiple Items dialog box displays, as shown here:

3. Click Format to display the String Display Format dialog box, which allows you to change the color, font, text style, and text placement of the set of form items.

To learn more about the String Display Format dialog box, see “Changing the format of a form item” on page 139.
Creating mnemonics for form items

A mnemonic defines a shortcut way to select a form item with the keyboard. The label can be on a Button, a Topic Box, or it can be Static Text that labels another form item.

To create a mnemonic for a form item, place one of the following characters before the letter for which you want to create a mnemonic:

- **A tilde (~):** Use a tilde when you plan to deliver the application on more than one platform. To include a tilde as part of the form item, type two tilde characters (~~).
- **An ampersand (&):** The ampersand does the same thing as the tilde but conforms to Windows usage.

(Indirectly) setting a mnemonic on an Entry Field: This is useful when you wish to enable an application user to quickly set focus to an Entry Field. First, set a mnemonic on a Static Text Field by placing an ampersand (&) character in the label text, right before the letter you wish to use as the underlined mnemonic letter. Next, use the Item Groups dialog box (opened from the Designer’s Form > Item Traversal menu option) to group the Static Text Field with an Entry Field (make sure that the Static Text Field is the first form item in the group and that the Entry Field is the second). Now, when the user selects the mnemonic using Alt+<the underlined letter>, focus goes straight to the grouped Entry Field.

Mnemonics vs. accelerators: Because they both involve keyboard shortcuts, it can be easy to confuse mnemonics and accelerators. A mnemonic is a keyboard shortcut, indicated by an underlined letter, that is activated by pressing ALT+the underlined letter. On the other hand, an accelerator is a keyboard shortcut, consisting of one key or a combination of keys, that executes a method. For information on accelerators, see “Defining accelerator key events” on page 311.

Changing the format of a form item

You can change the font, colors, justification, and numeric format for many form items. When you click Format on the options dialog box of a form item, the system opens the appropriate type of Format dialog box.

The sections that follow describe the following types of Format dialog boxes that can display:

- Date Display Format dialog box
- Numeric Display Format dialog box
- String Display Format dialog box
Date Display Format dialog box

The Date Display Format dialog box, as shown here, allows you to define display characteristics of a Date Field form item:

To use the dialog box, change any of the elements on the form. Click OK to exit and save your changes, or click Cancel to exit without saving your changes.
**Dialog box elements.** The following table describes the elements of the dialog box, presented alphabetically:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>Drop-down List Box of the available background colors.</td>
</tr>
<tr>
<td>Example</td>
<td>Sample of the format you have selected.</td>
</tr>
<tr>
<td>Font</td>
<td>Drop-down List Box of the available fonts.</td>
</tr>
<tr>
<td>Foreground</td>
<td>Drop-down List Box of the available foreground colors.</td>
</tr>
<tr>
<td>Horizontal justification</td>
<td>Radio Buttons that set the horizontal position of the date within the Date Field.</td>
</tr>
<tr>
<td>Period</td>
<td>Drop-down List Box of available periods (for example, Years). You must select Period from the Type Drop-down List Box.</td>
</tr>
<tr>
<td>Prefix</td>
<td>Text to display to the left of the date.</td>
</tr>
<tr>
<td>Separator label</td>
<td>The separator character to use with numeric date formats.</td>
</tr>
<tr>
<td>Size</td>
<td>Drop-down List Box of the available point sizes.</td>
</tr>
<tr>
<td>Style</td>
<td>Style of text to display (for example, italic).</td>
</tr>
<tr>
<td>Suffix</td>
<td>Text to display to the right of the date.</td>
</tr>
<tr>
<td>Type</td>
<td>Drop-down List Box of the available date formats (for example, MMDDYY).</td>
</tr>
<tr>
<td>Vertical justification</td>
<td>Radio Buttons that set the vertical position of the date within the Date Field.</td>
</tr>
</tbody>
</table>
**Numeric Display Format dialog box**

The Numeric Display Format dialog box, as shown here, allows you to define display characteristics of a form item that displays numbers:

To use the dialog box, change any of the elements on the form. Click **OK** to save your changes and exit, or click **Cancel** to exit without saving your changes.
**Dialog box elements.** The following table describes the elements of the dialog box, presented alphabetically:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>Drop-down List Box of the available background colors.</td>
</tr>
<tr>
<td>Country</td>
<td>Drop-down List Box of the available country formats. These formats affect the display of currency, separator, and decimal characters.</td>
</tr>
<tr>
<td>Enable leading</td>
<td>Valid when Enable width is checked and greater than zero. Places leading characters before the number in the field (enable width – number length).</td>
</tr>
<tr>
<td>Enable precision</td>
<td>Enables the amount of precision, in number of decimal places. Used for fixed and currency numbers.</td>
</tr>
<tr>
<td>Enable width</td>
<td>Number of characters in the field.</td>
</tr>
<tr>
<td>Example</td>
<td>Sample of the format you have selected.</td>
</tr>
<tr>
<td>Font</td>
<td>Drop-down List Box of the available fonts.</td>
</tr>
<tr>
<td>Foreground</td>
<td>Drop-down List Box of the available foreground colors.</td>
</tr>
<tr>
<td>Format with separators</td>
<td>Toggles whether to display the number with thousands separators appropriate to the country you selected.</td>
</tr>
<tr>
<td>Horizontal justification</td>
<td>Horizontal position of the number within the field.</td>
</tr>
<tr>
<td>Percentage</td>
<td>Toggles whether to display the number as a percentage.</td>
</tr>
<tr>
<td>Prefix</td>
<td>Text to display to the left of the number.</td>
</tr>
<tr>
<td>Size</td>
<td>Drop-down List Box of the available point sizes.</td>
</tr>
<tr>
<td>Style</td>
<td>Style of text to display (for example, italic).</td>
</tr>
<tr>
<td><strong>Element</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Suffix</td>
<td>Text to display to the right of the number.</td>
</tr>
<tr>
<td>Type</td>
<td>Drop-down List Box of the available number formats (described in the following table).</td>
</tr>
<tr>
<td>Vertical justification</td>
<td>Vertical position of the number within the field.</td>
</tr>
</tbody>
</table>

**Type.** The following table shows the effect of the choices in the Type drop-down list box:

<table>
<thead>
<tr>
<th><strong>Choice</strong></th>
<th><strong>Sample number</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>1234.56</td>
</tr>
<tr>
<td>Integer</td>
<td>1235</td>
</tr>
<tr>
<td>Fixed</td>
<td>1234.5600000000000</td>
</tr>
<tr>
<td>Scientific</td>
<td>1.2345600000000e+03</td>
</tr>
<tr>
<td>Currency</td>
<td>1234.56</td>
</tr>
</tbody>
</table>

**String Display Format dialog box**

The String Display Format dialog box, as shown here, allows you to define display characteristics of a form item that displays strings:

To use the dialog box, change any of the elements on the form. Click **OK** to save your changes and exit, or click **Cancel** to exit without saving your changes.
**Dialog box elements.** The following table describes the elements of the dialog box, presented alphabetically:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>Drop-down list box of the available background colors.</td>
</tr>
<tr>
<td>Example</td>
<td>Sample of the format you have selected.</td>
</tr>
<tr>
<td>Font</td>
<td>Drop-down List Box of the available fonts.</td>
</tr>
<tr>
<td>Foreground</td>
<td>Drop-down List Box of the available foreground colors.</td>
</tr>
<tr>
<td>Horizontal justification</td>
<td>Horizontal position of the string within the field.</td>
</tr>
<tr>
<td>Prefix</td>
<td>Text to display to the left of the string.</td>
</tr>
<tr>
<td>Size</td>
<td>Drop-down List Box of the available point sizes.</td>
</tr>
<tr>
<td>Style</td>
<td>Style of text to display (for example, italic).</td>
</tr>
<tr>
<td>Suffix</td>
<td>Text to display to the right of the string.</td>
</tr>
<tr>
<td>Vertical justification</td>
<td>Vertical position of the string within the field.</td>
</tr>
</tbody>
</table>
Declaring the source of data for a form item

The Link Hookups dialog box allows you to declare the source of data for a form item that is not an Interface Component. For example, you can declare that the source for a Tabular List Box is a table in an external database, or that the source for a Rectangle is a model that you have defined.

There are two types of links, as described in the following table:

<table>
<thead>
<tr>
<th>Link Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read link</td>
<td>Displays data but does not allow updating. If the data changes, the form item is updated to reflect the change.</td>
</tr>
<tr>
<td>Update link</td>
<td>Displays data and also allows user updates. When the data changes, the form item is updated to reflect the change. When the user changes the data displayed in the form item, the source is updated to reflect the change.</td>
</tr>
</tbody>
</table>
Using the Link Hookups dialog box

To use the Link Hookups dialog box:

1. Click Link on the options dialog box for the form item. Note that not all form items have a Link button.

   The Link Hookups dialog box displays, as shown here:

   ![](link_hookups.png)

   Select from each of the list boxes (proceeding from left to right).

   2. Select the type of object where you want to create a link. By default, you can select a Controller, a Model, a SqlSelect, or a Table (database). As you create new objects, other choices can display.

   3. Select the name of the object to use as a target. For example, if you select Table in step 1, choose from tables such as Accounts, Customer, and Employee.
4. Select the element that acts as the source. For example, if you selected
Orders in step 2, choose orderdate and Date (column and data type), as
shown here:

5. Click OK to save your changes and exit, click Cancel to exit without saving,
or click Clear to delete a previously defined link.
Example. To create a List Box containing employee names:

1. Add a List Box to a form.
2. Double-click the List Box to display the Form List Box Options dialog box.
3. Click Link to display the Link Hookups dialog box.
4. From the leftmost list box, select Table.
   A list of the available tables displays in the center list box on the Link Hookups dialog box.
5. From the center list box, select Employee.
   The system displays columns of the Employee table.
6. From the rightmost synchronized list box, select name.
7. Click OK.
   The system returns focus to the Form List Box Options dialog box.
8. Click OK.
   The system returns focus to the form you are working on.

To test the interface, select File > Test interface from the Designer menu. The List Box should be populated with employee names.

When the system loads the controller that owns a linked form item, it also loads the object to which the form item is linked.

Deleting a link
To delete a link:

1. Double-click the form item to display the options dialog box for the form item.
2. Click Link to display the Link Hookups dialog box.
3. Click Clear.
4. Click OK to close the Link Hookups dialog box.

Deleting an update link
To delete an update link:

1. Double-click the form item to display the Options dialog box.
2. In the list box next to the link buttons, select the Update link to delete.
3. Click Delete Link.
Assigning methods to form items

The Method Assignment dialog box, as shown here, allows you to associate methods with form item events:

Before you display the Method Assignment dialog box, ensure that the method you are assigning already exists for the controller, either as a system-provided method or as a method that you create in the Method Editor.

Note that the dialog box contains the form item to which you are assigning a method, next to the Item Name label. It also contains a list box of available methods.

If you are creating an interface or an interface component that is based on a subclass of a subclass of Controller or InterfaceComponent, and if you check the Inherited Methods check box, the list box contains all of the methods defined in the superclasses and in this class.

For more information about defining methods, see “Using the Method Editor to create a method” on page 61. For more information about the types of form item events, see “Form item events” on page 314.
Assigning a method
To assign a method to the form item:

1. Select the Event Type radio button that corresponds to the type of event for which you are defining a method.

2. Select the method that you want to assign from the Available Methods list box.

3. Click Copy.

   The method you selected displays in the Assigned Methods list box.

Deleting an assigned method
To delete an assigned method:

1. Select the method that you want to delete from the Assigned Methods list box.

2. Click Delete.

   The system removes the method from the Assigned Methods list box.

Exiting the Method Assignment dialog box
To exit the dialog box, click OK to save your changes or click Cancel to exit without saving your changes.
Setting form item attributes

The Form Item Options dialog box allows you to set form item attributes and to declare to which keys the form item responds. For different form items, the Form Item Options dialog box has some variations.

A typical example of the dialog box is shown here:

Check the Miscellaneous and Keys check boxes that you want to enable. The dialog box has two topic boxes, listed here and described in the following sections:

- **Miscellaneous**: The Miscellaneous section lists attributes of the form item that you can set.
- **Keys**: The Keys section lists the keys to which the form item can respond.
**Miscellaneous topic box**

The following table describes the attributes that can display in the *Miscellaneous* topic box of the Form Item Options dialog box, presented alphabetically:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning when checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow Drag</td>
<td>In a Tree View, allows the user to drag a label to a different position.</td>
</tr>
<tr>
<td>Allow Label Edit</td>
<td>In a Tree View, allows the user to change the text of the label.</td>
</tr>
<tr>
<td>AutoEdit</td>
<td>For a Workplace Object Template, adds a second way to create an object (the user can double-click the template). If unchecked, the user can create a new object by dragging from the template.</td>
</tr>
<tr>
<td>AutoSkip</td>
<td>When the user fills a form item’s field with text, the cursor skips to the next form item.</td>
</tr>
<tr>
<td>Bitmap</td>
<td>In a Draw Pad, displays bitmaps. If unchecked, you can write code that allows the user to draw in the Draw Pad.</td>
</tr>
<tr>
<td>Disallow Dragging</td>
<td>Prevents the user from dragging items out of a Value Set.</td>
</tr>
<tr>
<td>Display Only</td>
<td>Makes a Slider read only by removing the Slider arm.</td>
</tr>
<tr>
<td>Double click</td>
<td>In a Tree View, the user can expand or collapse nodes with a double-click or a click. If unchecked, the user can only expand or collapse nodes with a click.</td>
</tr>
<tr>
<td>Draw3D</td>
<td>Draws a shadow around a Topic Box so that the box appears to display in three dimensions. This feature works best with a gray background.</td>
</tr>
<tr>
<td>Dynamic Expand</td>
<td>In a Tree View, allows the user to create nodes dynamically (for example, creating children nodes when the parent is expanded).</td>
</tr>
<tr>
<td>EditFiles</td>
<td>Notifies a Text Editor that it is working with files, as opposed to text that stays in memory. Includes file activities in the Text Editor such as saving to a file and making backups.</td>
</tr>
<tr>
<td>Enable Mouse</td>
<td>Allows the form item to respond to mouse clicks (for example, on a Bar Chart).</td>
</tr>
<tr>
<td>Extended Selection</td>
<td>Allows the user to select more than one item in a sequence by either dragging the mouse from the first item to the last or by selecting the first item, pressing the <strong>SHIFT</strong> key, and selecting the last item.</td>
</tr>
<tr>
<td>Grouped Bars</td>
<td>Groups multiple values in a Bar Chart.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Meaning when checked</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Help Available</td>
<td>Allows you to assign the functionality of the F1 key to a Button. When the user presses F1, help about the form item that has focus displays.</td>
</tr>
<tr>
<td>HSlider</td>
<td>For a Slider, makes the orientation horizontal. If unchecked, makes the orientation vertical.</td>
</tr>
<tr>
<td>Horz. Moveable</td>
<td>When the form is resized, the system maintains a fixed distance between the right side of the form and the right side of the form item. If this option is checked, you cannot check Horz. Resizeable.</td>
</tr>
<tr>
<td>Horz. Resizeable</td>
<td>When the form is resized, the system resizes the form item horizontally to maintain a fixed ratio between the width of the form and the width of the form item. If this option is checked, you cannot check Horz. Moveable.</td>
</tr>
<tr>
<td>Horz. Scroll Bar</td>
<td>Assigns a horizontal scroll bar to the form item.</td>
</tr>
<tr>
<td>Line Graph</td>
<td>Creates a Line Graph instead of a Bar Chart.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Allows you to create a mnemonic for Static Text or a Topic Box.</td>
</tr>
<tr>
<td>MultiLine</td>
<td>Allows the user to enter more than one line of text in the form item.</td>
</tr>
<tr>
<td>Multiple Selection</td>
<td>Allows the user to select more than one item from a list by pressing the CTRL key when selecting the second and subsequent item or by dragging the mouse from the first item to the last.</td>
</tr>
<tr>
<td>No Border</td>
<td>Removes or replaces the border around the form item.</td>
</tr>
<tr>
<td>Not Editable</td>
<td>Prevents the user from editing the information in the form item, making the form item read only.</td>
</tr>
<tr>
<td>Not Selectable</td>
<td>Prevents the user from setting focus to the form item (and thus from editing the information in the form item).</td>
</tr>
<tr>
<td>No Vert. Scroll Bar</td>
<td>Removes the vertical scroll bar from the item.</td>
</tr>
<tr>
<td>No Word Wrap</td>
<td>A line of text that exceeds the width of a Static Text box does not wrap to the next line. If unchecked, the system wraps text that is too long for the Static Text box.</td>
</tr>
<tr>
<td>Only Numbers</td>
<td>Restricts the display in a Spin Button to numeric values, and right justifies them.</td>
</tr>
<tr>
<td>Password</td>
<td>Prevents the system from displaying data associated with the form item. Instead, the system displays an asterisk (*) for each character that the user types in the item.</td>
</tr>
<tr>
<td>Pie Chart</td>
<td>Creates a Pie Chart instead of a Bar Chart.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Meaning when checked</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Reduced Tree</td>
<td>In a Tree View, shows only the direct line of inheritance in an expanded tree.</td>
</tr>
<tr>
<td>Show Dots</td>
<td>In a Tree View, displays an ellipsis next to the name of a node that can be expanded (for example, Text...).</td>
</tr>
<tr>
<td>Show Icon</td>
<td>In a Tree View, the system displays the icons of the expanded nodes and the collapsed nodes next to nodes in the tree. If unchecked, the system does not display icons next to the nodes.</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>Customizes a Text Editor to work with Smalltalk code rather than ASCII text. For example, allows you to color code Smalltalk code.</td>
</tr>
<tr>
<td>Stacked Bars</td>
<td>Stacks multiple values in a Bar Chart.</td>
</tr>
<tr>
<td>Track Mouse</td>
<td>Allows the form item to respond to the cursor passing over the item, even if the user does not press a mouse button.</td>
</tr>
<tr>
<td>Vert. Moveable</td>
<td>When the form is resized, the system maintains a fixed distance between the bottom of the form and the bottom of the form item. If this option is checked, you cannot check Vert. Resizeable.</td>
</tr>
<tr>
<td>Vert. Resizeable</td>
<td>When the form is resized, the system resizes the form item vertically to maintain a fixed ratio between the height of the form and the height of the form item. If this option is checked, you cannot check Vert. Moveable.</td>
</tr>
<tr>
<td>Vert. Scroll Bar</td>
<td>Assigns a vertical scroll bar to the form item.</td>
</tr>
<tr>
<td>Word Wrap</td>
<td>When the user types more text than can fit on one line, the system breaks the line and starts a new line.</td>
</tr>
</tbody>
</table>

**Keys topic box**

The Keys topic box contains a list of the nonprinting keys (for example, the Esc key) that can be enabled for the form item. When a check box in the Keys topic box is checked, it means that the form item can respond when the user presses that key.

Once you define a key press to which the form item can respond, use the Method Assignment dialog box to create a hit key event for the key. For more information about hit key events, see “Form item events” on page 314.
Moveable Borders

You can provide horizontal and vertical Moveable Borders for the items on your forms. These borders allow the user to resize lists, Tree Views, text fields, or any other item. An example window, before and after resizing, is shown here:

When the user places the cursor on a horizontal or vertical moveable border, the cursor will change into a two-sided arrow. In this state, the user can start dragging the border to resize the item. You control how far the user is allowed to drag the border and how the items will get resized as a reply to this drag.
Creating Moveable Borders

You can create interfaces with Moveable Borders entirely from the ObjectStudio Designer without any method coding. The necessary resizes and moves on the visible items can be visibly programmed by using events.

To move or resize items, use the positionChangedBy: event of the vertical or horizontal Border in the Event Editor window, as shown here:

With this event send one of the following inherited messages to the item you want to reposition:

- resizeBy: or moveBy: if the border is to the right of or below the item
- invResizeBy:, invMoveBy, or moveAndShrinkBy: if the border is to the left of or above the item
Moveable Border options

You can also set options to configure the borders, as shown here for a Vertical Border:
Below are explanations of the elements of the Form Vertical Border Options dialog box:

<table>
<thead>
<tr>
<th>Entry Field</th>
<th>Allows you to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Enter the name of the Moveable Border for which the options will apply.</td>
</tr>
<tr>
<td>Background Color</td>
<td>Specify a color for the border. If you specify Transparent, the border becomes the same color as the forms background, which makes it seem invisible. To eliminate the border, click the Options button and check NoBorder in the dialog box.</td>
</tr>
<tr>
<td>Store position</td>
<td>Make the ratio persistent, so that when the user reopens the window it will look the same as when it was last closed. Check Store position so that the ratio will be written in the ostudio.ini file in the user’s home directory. This option is available only when Keep Ratio is checked.</td>
</tr>
<tr>
<td>Keep Ratio</td>
<td>Maintain the proportions of an item when the user resizes the window. This option will cause the borders of the item to move in proportion to the size of the window. With this option you can expand ObjectStudio's default behavior of FormItems for vertical and horizontal resizeability.</td>
</tr>
<tr>
<td>Border Distances</td>
<td>Limit how close to the window borders the user can drag a Moveable Border. Enter Border Distance values in increments of 1/10 millimeters. The user will not be able to drag a Moveable Border closer than this distance from the window border.</td>
</tr>
</tbody>
</table>
Enabling or disabling smart list search and smart list sounds

Enabling or disabling smart list search

For characteristics of smart list search, see “Characteristics of smart list search” on page 162.

You can toggle smart list search off or on for all ObjectStudio lists except for Tabular List Boxes; Tabular List Boxes always have smart list search enabled. ObjectStudio list form items are the following:

♦ Combo List
♦ Drop-down Combo Box
♦ Drop-down List Box
♦ List Box
♦ ListView
♦ Tabular List Box
To enable or disable smart list search, open the Desktop Options window by selecting **View > Options** from the Desktop menu, and select or deselect the **disable smart list search** option.

Setting this option affects all lists in ObjectStudio except for tabular list boxes; tabular list boxes always have smart list search enabled. You cannot disable or enable smart list search for individual lists.
Enabling or disabling smart list sounds
You can toggle smart the smart list beep sound for all lists in ObjectStudio (ListView, List Box, Tabular List, etc.). To set this option, open the Desktop Options window by selecting View > Options from the Desktop menu, and select or deselect the disable click sound option.

Description of smart list sounds: If the disable click sound option is deselected, when you start typing letters in order to select an item from a selected smart list, a short, high-pitched beep occurs when the last typed letter matches a list item and a short, lower-pitched beep occurs when the last typed letter fails to match a list item. If the disable click sound option is selected, no beeps occur as a result of using a smart list.

Setting this option affects all lists in ObjectStudio. You cannot disable or enable smart list sounds for individual lists.

Characteristics of smart list search
Smart list search has the following behaviors:
♦ If a list has focus and the user presses a character, the first list item that starts with that character is selected.
♦ If the selected item is not visible, the list will scroll it into view.
♦ If there are no matches found searching down in the list from the current position, the search will continue from the top of the list.
♦ The search starts immediately after the user presses a character. If the user presses another character within a time interval defined in OnCharListSearcher(class)\textgreater defaultInitialValueForTimeDelay, the character will be appended to the previous search buffer. The current search will then be canceled and a new search started.

The default time interval is 1.5 seconds. This interval can be changed at any time in ObjectStudio. This is different from Windows list behavior where it is not possible to change the time interval.
♦ If the list loses focus after a search, the search will continue from the last position after focus is regained. If, however, the user presses a character while another list has focus, the search in the first list will be cancelled.
♦ For ListView and TabularLists, the search will be conducted in the specific column that the user clicks with the mouse pointer.
♦ No matter how the disable smart list search option is set in the Desktop Options dialog box, Tabular List Boxes always use smart list search.
Form item descriptions

This section presents information about each form item. Information is arranged alphabetically. Each description contains:

♦ The form item’s icon as it displays in the Designer’s New Item dialog box
♦ A description of the form item
♦ Where appropriate, information about using the Designer to customize the form item

Bitmap Selector

A Bitmap Selector is an interface component that allows the user to select a system icon, a system bitmap, or a bitmap file to insert in the application. The Bitmap Selector form item is shown here:

When you use a Bitmap Selector component, you need to configure the controller so that it will set the Bitmap Selector’s initial selection. For example, you can specify that the initial selection will be System Icon. The following sample code shows how you can set the initial selection to System Icon for a Bitmap Selector named “cBitmap”:

```plaintext
postOpenInitialization
    cBitmap put: (Array with: #SystemIcon with: nil).
```
Note that the Bitmap Selector has these features:

♦ Image Type radio buttons, from which you can select the type of bitmap to insert

♦ A Spec drop-down list box or a Select File button that you can use to specify the bitmap

♦ An Icon size drop-down list box that you can use to specify the size of an icon if you select the File radio button from among the Image Type radio buttons

♦ An Example area that displays the bitmap you have selected

To learn more about interface components, see “Interfaces and interface components” on page 246.
Button

A Button is an object that the user can click to initiate an action. On many interfaces, it is common to include an OK button and a Cancel button.

Label. The Button’s label can be text or a bitmap. Change the label by changing settings in the Button Type radio button box on the Form Button Options dialog box. The following table describes the Button Type options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Indicates that the label is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Text. Enter the text in the Text entry field.</td>
</tr>
<tr>
<td>Auto Bitmap</td>
<td>A bitmap or icon that is either stored in the system or in a file. When you select this option, the system displays a Bitmap Selector that allows you to specify the bitmap or icon.</td>
</tr>
<tr>
<td>User Bitmap</td>
<td>A bitmap or icon that can change according to the Button’s state. When you select this option, the system displays a Bitmap Selector that allows you to specify the bitmap or icon. It also displays a User Defined radio button box that allows you to associate the bitmap or icon with a Button state.</td>
</tr>
</tbody>
</table>

You cannot change the foreground and background colors of a Button.

Methods. The Form Button Options dialog box allows you to associate a method with a Button. The method executes when the user clicks the Button. You can write the method or use a predefined one. Some of the predefined methods open or close forms or display the next or previous record.

Defining a mnemonic for a Button

You can create a mnemonic for a Button so that the user can use the keyboard to activate the Button.

For example, to create a mnemonic for the Button label, Modify, type one of the following into the Text entry field on the Form Button Options dialog box:

- `Mo~dify`
- `Mo&dify`

To use the mnemonic in this example:

1. Set focus to the form that contains the Modify button.
2. Press ALT+D.

The system responds as if the Modify button had been clicked.
Defining Default and Escape Buttons

When a Default Button is defined for a form, the user can set focus to the form and press the ENTER key. The system responds as if the user had clicked the Button designated as the Default. The Default Button action will continue to be triggered by the ENTER key until the user tabs to a different Button, thereby setting focus to that Button. When the user does this and then presses the ENTER key, the system responds as if the user had clicked the Button that has focus. When the form is closed and then reopened, pressing the ENTER key again triggers the Default Button action. If the user tabs to a non-Button form item, thereby setting focus to that form item, pressing the ENTER key triggers the Default Button action.

When an Escape Button is defined for a form, the user can set focus to the form and press the Esc key. The system responds as if the user had pressed the Button designated as the Cancel button. (Typically, when the user clicks the Cancel button, the application cancels unsaved work and closes the window.)

To define a Default Button or an Escape Button, check the appropriate Check Box on the Form Button Options dialog box. Defining a Default Button for a form overrides the definition for a Return key method.

Return key method. You can make a Button the Default Button so that pressing ENTER has the same effect as clicking the Button. To create a Default Button:

1. Create a return key method for the Button. For more information, see “Defining return key events” on page 313.

2. Send the highlightOn message to the instance of ButtonCtrl associated with the Button. The highlightOn method highlights the Button so that the user knows which Button is the default.

To remove highlighting from a Button, send the highlightOff message.
Chart

A Chart displays numeric information graphically. You can display the information as a:

♦ Bar Chart (the default)
♦ Line Graph
♦ Stacked Bars
♦ Grouped Bars
♦ Pie Chart

You can add static or dynamic labels to a Chart, and you can enter the minimum and maximum number of entries to display.

**Changing the format**

By default, a Chart is displayed as a Bar Chart.

To change to another format:

1. Double-click the Chart form item to display the Form Chart Options dialog box.
2. Click **Options** to display the Form Item Options dialog box.
3. Check the appropriate check box to change the format of the Chart.
4. Click **OK** on each dialog box to return to the Designer.

**Mouse options.** The following table describes the options on the Form Item Options dialog box that are related to the mouse:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning when checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Mouse</td>
<td>Allows the user to select an item from the Chart.</td>
</tr>
<tr>
<td>Track Mouse</td>
<td>Allows the system to track the mouse as it moves over the Chart.</td>
</tr>
</tbody>
</table>
Adding data to a Chart

To add data to a Chart:

1. Add a Chart form item to the form. You can resize the Chart; for example, double its original height and width.

2. Double-click the Chart form item to display the Form Chart Options dialog box.

3. Link the Chart to a data source:
   A. Click Link to display the Link Hookups dialog box.
   B. From left to right, select Table, Employee, and salary (or any numeric field).
   C. Click OK to close the dialog box.

4. Edit the Chart:
   A. Click Options to display the Form Item Options dialog box.
   B. Check the Enable Mouse and Track Mouse check boxes.
   C. Click OK to close the dialog box.

5. From the Color drop-down list box, select the color of the bars in the Chart. For example, select Blue.

6. From the Selection color drop-down list box, select the color of selections in the Chart. For example, select Red.

7. Check the Enable Display Minimum check box and type a number in the Entry Field to set the minimum number of entries to display. For example, enter 5.

8. Check the Enable Display Maximum check box and type a number in the Entry Field to set the maximum number of entries to display. For example, enter 50.

9. Click OK to exit the dialog box.

10. Save and test the interface.

The system displays a Bar Chart of the data you selected. When you click one of the bars in the Chart, the system changes its color to the Selection color you chose in step 6.
**Indicating or changing the position in the data source**

You can use a Chart to indicate the current position in the linked data source (for example, the current row in a table). You also can use a Chart to change the current position in the linked data source.

To change the position in the data source:

1. Add a Chart form item to the form.
2. Link it to a data source.
3. On the Form Item Options dialog box, check the **Enable Mouse** and **Track Mouse** check boxes.

When you select a bar in the Chart, the system resets the current position in the data source. To test this functionality:

1. Create the Chart described in the previous example.
2. Add an Entry Field to the form.
3. Double-click the Entry Field to display the Form Entry Field Options dialog box.
4. Click **Link** to display the Link Hookups dialog box. From left to right, select **Table**, **Employee**, and **name**. Click **OK**.
5. Click **OK** to close the Form Entry Field Options dialog box.
6. Save and test the interface.

The Chart should have one selected bar and a name should display in the Entry Field, indicating the current position in the Employee table. If you select another bar in the Chart, the name in the Entry Field changes to indicate a new position in the Employee table.
Adding labels to a Chart

There are two kinds of labels:

♦ **Static:** Static labels reflect the text that you enter in the Labels and Legends dialog box.

♦ **Dynamic:** Dynamic labels are created by defining a link (for example, a link to a table).

To add a label to a Chart:

1. Double-click the Chart form item to open the Form Chart Options dialog box.
2. Select the **Static Labels** or **Dynamic Labels** radio button.
3. Click **Edit Labels**.
4. Type the first label in the entry field on the Labels and Legends dialog box.
5. Click **Add**.
6. Repeat steps 4 and 5 for each label.
7. Click **OK** to close the Labels and Legends dialog box.
8. Click **OK** to close the Form Chart Options dialog box.
9. Save and test the interface.
Adding a legend to a Chart

To add a legend to a chart:

1. Double-click the Chart form item to open the Form Chart Options dialog box.
2. Click **Legend**.
   
   A legend box displays on the form.
3. Click **OK** to close the Form Chart Options dialog box.
4. Double-click the legend box to make changes to the labels on the legend.
   
   The Labels and Legends dialog box displays, as shown here: You can add, change, or delete labels.

5. Click **Options** to display the Form Item Options dialog box. Make changes if needed.
6. Click **OK** to close the Labels and Legends dialog box.
7. Save and test the interface.
Check Box

A Check Box presents the user with a choice. The user checks the Check Box to indicate a value of On or True. The user unchecks the Check Box to indicate a value of Off or False.

Display-only Check Boxes

You can create a Check Box that is read only. For example, you can display the result of a choice made elsewhere in the application but not allow the user to change the choice. To do so, check the Not Editable or Not Selectable check box on the Form Item Options dialog box. To learn more about these options, see “Setting form item attributes” on page 152.
Check List

A Check List is a collection of logically related Check Boxes (see “Check Box,” above). The user can select (check) zero or more choices in the Check List. For example, when you format text, it is typical to use a Check List for the text style, as illustrated here:

Display-only Check Lists

You can create a Check List that is read only. For example, you can display the result of a choice made elsewhere in the application but not allow the user to change the choice. To do so, check the Not Editable or Not Selectable check box on the Form Item Options dialog box. To learn more about these options, see “Setting form item attributes” on page 152.

Adding items to a Check List

To create the Check List shown in on the previous page:

1. Add a Check List form item to the form.
2. Double-click the newly added Check List form item to display the Form Check List Options dialog box.
3. In the entry field under the Labels list box, type (press ENTER or click Add after each entry to add the label to the Labels list box):
   - Bold
   - Italic
   - Underscore
   - Strikethrough
4. Click OK to return to the Designer.

The Check List with the items that you just added displays. You may need to resize the form item if any of the labels are truncated.
Combo List

A Combo List is a combination of an Entry Field and a List Box. For more information, see “Entry Field” on page 187 and “List Box” on page 190. The user can select an item from the List Box or, in the Entry Field, type a choice that is not in the List Box.

The ControllerItem and its FormItem can have different selections. If the value in the entry field does not precisely match an item in the list, the ControllerItem selection will be zero and the FormItem selection will be the number of the item selected in the list.

Adding list items

Using the Designer, there are two ways to add list items to a Combo List:

♦ Enter items manually. To enter list items manually:
  1. Double-click the Combo List form item to display the Form Combo List Options dialog box.
  2. In the entry field under the Labels list box, type the name of each list item.
  3. Press ENTER or click Add to add the list item to the Labels list box.
  4. Click OK to save your work and close the dialog box.

♦ Use the Link facility to declare a data source. For more information about the Link facility, see “Declaring the source of data for a form item” on page 146.

Using smart list search

A Combo List can use smart list search. Smart list search allows the user to quickly find an entry in a list by typing the first few characters of the entry in the list form item. If you would like to enable or disable smart list search, see “Enabling or disabling smart list search” on page 160.
Date Field

A Date Field is an Entry Field that allows you to format the display of dates and verify the entry of date values (see “Entry Field” on page 187).

When the user enters text into the field and sets focus to another form item, the system determines whether the text is a date. If it is a date, the system reformats the text according to the format you associate with the field. If it is not a date, the system beeps.

If you use the put: method to place data in a Date Field, the system does not format or validate the data. Therefore, if you add data to a Date Field programmatically, ensure that the data is valid and formatted correctly.

To learn more about specifying a date format, see “Changing the format of a form item” on page 139. To learn more about how the system validates data and how to change the system’s default validation behavior, see “Validating data” on page 357.

Draw Pad

A Draw Pad is an area where:

♦ A bitmap can be displayed
♦ A user can draw

Mouse options. The following table describes the options on the Form Item Options dialog box that are related to the mouse:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning when checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Mouse</td>
<td>Allows the user to select an item from the Draw Pad.</td>
</tr>
<tr>
<td>Track Mouse</td>
<td>Allows the system to track the mouse as it moves over the Draw Pad.</td>
</tr>
</tbody>
</table>
Initializing a Draw Pad with a bitmap

To load a bitmap into a Draw Pad when the Draw Pad is created:

1. Add a Draw Pad form item to the form.
2. Double-click the Draw Pad to display the Form Draw Pad Options dialog box.
3. Edit the bitmap:
   A. Click Options to display the Form Item Options dialog box.
   B. Check the Bitmap check box.
   C. Click OK.
4. Click OK to close the Form Draw Pad Options dialog box.
5. Select Tools > Methods from the Designer menu to open the Method Editor.
6. Create the following method:
   A. Click New on the Method Editor.
   B. In the Source edit area, type (substitute the code in italics where appropriate):

```plaintext
openInitialization
    bmp := Bitmap loadFile: 'c:\mybmp.bmp'.
drawPad1 put: bmp.
```
   C. Click Save to save the method.
      You may be prompted to subclass the method. Click OK to subclass the method.
   D. Click Close to close the Method Editor.
7. Save and test the interface.

The bitmap displays in the Draw Pad.
Loading a bitmap dynamically

This section shows you how to load a bitmap into a Draw Pad dynamically. It requires two steps. First, you load the bitmap into memory; then you load it into the Draw Pad.

One advantage to this approach is that after you load the bitmap into memory, the application does not need access to the bitmap file.

To load a bitmap dynamically:

1. Select **Tools > Methods** from the Designer menu to open the **Method Editor**.

2. Create the following method:
   A. Click **New** on the **Method Editor**.
   B. In the **Source** edit area, type (substitute the code in italics where appropriate):

      ```
      finishInit
      | bmp |
      bmp := Bitmap loadFile: 'c:\mybmp.bmp'.
      bmp registerAs: #mybmp.
      ```
   C. Click **Save** to save the method.
      You may be prompted to subclass the method. If so, click **OK** to subclass the method.

3. Create the following method:
   A. Click **New** on the **Method Editor**.
   B. In the **Source** edit area, type:

      ```
      loadBitmap
      drawPad1 put: #mybmp.
      ```
      This method places the loaded bitmap into the Draw Pad.
   C. Click **Save** to save the method.
   D. Click **Close** to close the **Method Editor**.

4. Add a Button to the interface.

5. Double-click the Button form item to open the Form Button Options dialog box.
6. Click **Methods** to open the Method Assignment dialog box.
7. Select the loadBitmap method from the **Available Methods** list box.
8. Click **Copy** to add the method to the **Assigned Methods** list box.
9. Click **OK** to close the Method Assignment dialog box.
10. Save and test the interface.

When you click the Button, the bitmap displays in the Draw Pad.

**Saving the contents of a Draw Pad**

To save the contents of a Draw Pad to a file:

1. Select **Tools > Methods** from the Designer menu to open the **Method Editor**.
2. Create the following method:
   A. Click **New** on the Method Editor.
   B. In the **Source** edit area, type:

```
saveToFile
  ^ (drawPad1 formItem drawPort)
  saveFile: 'c:\testbmp.bmp'.
```
   C. Click **Save** to save the method.
   D. Click **Close** to close the Method Editor.
3. Add a Button to the interface.
4. Double-click the Button form item to open the Form Button Options dialog box.
5. Click Methods to open the Method Assignment dialog box.
6. Select the saveToFile method from the **Available Methods** list box.
7. Click **Copy** to add the method to the **Assigned Methods** list box.
8. Click **OK** to close the Method Assignment dialog box.
9. Save and test the interface.
Creating a Draw Pad example

You can use the Draw Pad form item to create an interface where the user can draw lines, rectangles, and circles and choose the color of the item. The following figure shows an example of using the Draw Pad and radio button form items to create the interface:
Form items. To create the Draw Pad form item:

1. Select **Form > Change form** from the Designer menu.
   The Change Form dialog box displays.

2. Type new names in the **Form Name** and **Form Title** entry fields (for example, type Draw Pad Example in both fields).

3. Click **OK** to close the Change Form dialog box.

4. Select **FormItem > New Item** to open the New Item dialog box.

5. Create a Draw Pad form item on the main form. You can resize the form item so that it fills part of the form. Leave enough room for the Radio Buttons on the right side of the form.

6. Double-click the Draw Pad form item to display the Form Draw Pad Options dialog box.

7. Do the following on the Draw Pad Options dialog box:
   A. Leave “drawPad1” in the **Name** entry field.
   B. Click **Options** to open the Form Item Options dialog box. Check the following check boxes:
      ♦ **Enable Mouse**
      ♦ **Horz. Resizeable**
      ♦ **Vert. Resizeable**
   C. Click **OK**.
To create the radio button form items:

1. Place a Radio Buttons form item next to the Draw Pad form item.

2. Double-click the Radio Buttons form item to display the Form Radio Buttons Options dialog box.

3. Fill in the Form Radio Buttons Options dialog box:
   A. Type the following in the Name entry field:

   function

   B. Click Options to open the Form Item Options dialog box. Select the Horz. Moveable check box. Click OK.

   C. Type the following, one at a time, in the entry field under the Labels list box. Press Enter to add each label to the Labels list box.

   - Line
   - Rectangle
   - Circle

   D. Click OK.


5. Double-click the new Radio Buttons form item to display the Form Radio Buttons Options dialog box.
6. Fill in the Form Radio Buttons Options dialog box:
   A. Type color in the **Name** entry field.
   B. Click **Options** to open the Form Item Options dialog box. Select the **Horz. Moveable** check box. Click **Close**.
   C. Type the following, one at a time, in the entry field under the **Labels** list box. Press Enter to add each label to the **Labels** list box.
      ♦ Red
      ♦ Green
      ♦ Blue
      ♦ Black
      ♦ Cyan
      ♦ Yellow
      ♦ Pink
   D. Click **OK**.

You have finished creating the form items for this example. Now you need to create an instance variable and methods as described in the following procedures.

**Instance variable.** You need to create an instance variable that stores the coordinates where mouse button 1 is first pressed on the Draw Pad.

To create the instance variable:

1. Select **Tools > Variables** from the Designer menu to open the Instance Variables dialog box.
2. Click **Add** to open the String Dialog box.
3. Type `startPt` in the **Enter instance variable** entry field.
4. Click **OK** to close the String Dialog box.
5. Click **Close** to close the Instance Variables dialog box.
Methods: The following procedures describe the methods you need to create so you can draw in the Draw Pad. When you first click the mouse in the Draw Pad, the coordinates of that point are saved. You then can press mouse button 1 and drag the mouse to create the specified shape in the specified color.

The Enable Mouse option attaches methods that let you click and drag the mouse on the Draw Pad. The method called when you first press mouse button 1 is button1DownAt:.

To create a method that draws the selected graphical item in the Draw Pad:
1. Select Tools > Methods from the Designer menu to open the Method Editor.
2. Create a method that stores the point at which mouse button 1 is pressed:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:
      
      ```smalltalk
      button1DownAt: aPt
      startPt := aPt.
      ```
   C. Click Save.
3. Create a method that draws the shape after mouse button 1 is released:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:
      
      ```smalltalk
      button1UpAt: pt
      | radius x1 x2 y1 y2 |

      (function getSelection) == 1 ifTrue: [
      drawPad1 drawFrom: startPt to: pt.
      ].
      (function getSelection) == 2 ifTrue: [
      drawPad1 rectangleFrom: startPt to: pt.
      ].
      (function getSelection) == 3 ifTrue: [
      x1 := startPt x.
      x2 := pt x.
      y1 := startPt y.
      y2 := pt y.
      radius := (((x2 - x1) * (x2 - x1)) + ((y2 - y1) * (y2 - y1))) sqrt.
      drawPad1 circleAt: startPt radius: (radius rounded).
      ].
      drawPad1 refresh.
      startPt := Nil.
      ```
   C. Click Save.
4. Create a method that sets the colors for the color radio buttons:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:
      
      ```
      defineColor
      drawPad1 setPenColor: (color getValue asSymbol).
      ```
   C. Click Save.
5. Create a method that specifies the default color when the window is opened:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:
      
      ```
      openInitialization
      self defineColor.
      ```
   C. Click Save.
6. Click Close to close the Method Editor.
7. Double-click the color radio buttons form item to open the Form Radio Buttons Options dialog box.
8. Fill in the Form Radio Button Options dialog box:
   A. Click Methods.
   B. Select the defineColor method from the Available Methods list box.
   C. Click Copy to add the method to the Assigned Methods list box.
   D. Click OK to close the Method Assignment dialog box.
9. Click OK to close the Form Radio Buttons Options dialog box.

**Testing and saving the interface**

To save and test the interface:

1. Select File > Save as from the Designer menu.
2. Select a folder in which to save the file. Type drawpad.cls in the File name entry field.
3. Click Save to save the file and close the Save As dialog box.
4. Click Toggle test/edit on the toolbar.
5. Select a shape and color from the shape and color radio buttons. Press mouse button 1 and drag the mouse on the Draw Pad to create the shape.
Drop-down Combo Box

A Drop-down Combo Box is a combination of a Drop-down List Box and an Entry Field. The user can select an item from the Drop-down List Box, or, in the Entry Field, type a choice that is not in the Drop-down List Box.

For more information, see “Drop-down List Box” on page 186 and “Entry Field” on page 187.

The ControllerItem and its FormItem can have different selections. If the value in the entry field does not precisely match an item in the list, the ControllerItem selection will be zero and the FormItem selection will be the number of the item selected in the list.

Adding list items

Using the Designer, there are two ways to add list items to a Drop-down Combo Box:

♦ Enter items manually. To enter list items manually:
1. Double-click the Drop-down Combo Box form item to display the Form Drop-down Combo Box Options dialog box.
2. Type the name of the first list item in the entry field under the Labels list box.
3. Click Add or press ENTER to add the list item to the Labels list box.
4. Repeat steps 2 and 3 for each list item that you want to add.
5. Click OK to save your work and close the dialog box.

♦ Use the Link facility to declare a data source. For more information about the Link facility, see “Declaring the source of data for a form item” on page 146.

Using smart list search

A Drop-down Combo Box can use smart list search. Smart list search allows the user to quickly find an entry in a list by typing the first few characters of the entry in the list form item. If you would like to enable or disable smart list search, see “Enabling or disabling smart list search” on page 160.
Drop-down List Box

A Drop-down List Box is a space-saving List Box with two states, closed and open:

♦ When closed, it displays one line of text and an arrow.
♦ When open (the user presses the arrow button or presses ALT and the down arrow), it displays the list of choices in addition to the components displayed when it is closed.

Adding list items

Using the Designer, there are two ways to add list items to a Drop-down List Box:

♦ **Enter items manually.** To enter list items manually:
  1. Double-click the Drop-down List Box form item to display the Form Drop-down List Box Options dialog box.
  2. Type the name of the first list item in the entry field under the **Labels** list box.
  3. Click **Add** or press **ENTER** to add the list item to the **Labels** list box.
  4. Repeat steps 2 and 3 for each list item that you want to add.
  5. Click **OK** to save your work and close the dialog box.

♦ **Use the Link facility to declare a data source.** For more information about the Link facility, see “Declaring the source of data for a form item” on page 146.

If you programmatically disable a Drop-down List Box and close the application while it is disabled, the next time you run the application, the Drop-down List Box displays disabled but acts enabled. To avoid this problem, set enablement to True.

Using smart list search

A Drop-down List Box can use smart list search. Smart list search allows the user to quickly find an entry in a list by typing the first few characters of the entry in the list form item. If you would like to enable or disable smart list search, see “Enabling or disabling smart list search” on page 160.
Entry Field

An Entry Field is a one-line area that displays text and accepts user input. You also can create a Multi Line Entry Field using the Multi Line Entry Field form item (see “Multi Line Entry Field” on page 208).

Creating a password Entry Field

You can disable the display of text in an Entry Field (for example, so that the user can enter a password).

To create a password Entry Field:

1. Add an Entry Field form item to the form.
2. Double-click the Entry Field form item to display the Form Entry Field Options dialog box.
3. Click Options to display the Form Item Options dialog box.
4. Check the Password check box.
5. Click OK on both Options dialog boxes to return to the Designer.

If you redisplay the Options dialog box for the Entry Field, note that it has a new title: Form FormPassword Options.
Creating a mask

A mask specifies the types of characters and format of each character that the user can type into the Entry Field.

To create a mask:

1. Double-click the Entry Field form item to display the Form Entry Field Options dialog box.

2. In the Mask entry field, type a combination of literal characters and the characters listed in the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Allows</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Any character (forces all letters to be uppercase)</td>
</tr>
<tr>
<td>@</td>
<td>Any character</td>
</tr>
<tr>
<td>#</td>
<td>Any digit</td>
</tr>
<tr>
<td>ã</td>
<td>An alphabetic character (forces all letters to be uppercase)</td>
</tr>
<tr>
<td>?</td>
<td>An alphabetic character</td>
</tr>
</tbody>
</table>

The number of characters in the mask determines the number of characters that the user must enter. Precede any of the special characters with a semicolon (;) to enter any of these characters as literals.

3. By default, the system displays a space for each character in the Entry Field that the user needs to type. You can change the character, for example, to a question mark (?). To change the character, enter a new character in the Fill Character entry field.
4. If you would like the system to beep when the user enters a character that does not match the mask, check the **Beep on errors** check box.

Make certain that the Windows Default sound is set to a value other than None, or no beep will sound. Alter this setting by using the Sounds control panel.

5. Click **OK** to return to the Designer.

If you redisplay the Options dialog box for the Entry Field, it will have a new title: **Form FormMask Field Options**.

**Mask examples.** The following table describes different masks:

<table>
<thead>
<tr>
<th>To create a mask for</th>
<th>Use this mask code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two letters followed by two numbers</td>
<td>??##</td>
</tr>
<tr>
<td>A U.S. phone number</td>
<td>(###) ###-####</td>
</tr>
<tr>
<td>A U.S. social security number</td>
<td>####-####-####</td>
</tr>
</tbody>
</table>

In the Entry Field, the U.S. phone number displays as follows (where Δ represents a space):

(Δ Δ Δ) Δ Δ Δ Δ -Δ Δ Δ Δ

Note that four spaces follow the right parenthesis (the mask contains a literal space immediately following the right parenthesis).

To learn more about how the system validates data and how to change the system’s default validation behavior, see “Validating data” on page 357.
Horizontal Border

A Horizontal Border is a horizontal moveable border that enables a user to resize lists, Tree Views, text fields, or other items. For more information, see “Moveable Borders” on page 156.

Line

A Line is a graphical device that separates a form into sections or adds visual emphasis to a form. You can specify the orientation, width, color, and corner style of the Line.

List Box

A List Box is a box that has an optional scroll bar and contains items on separate lines. The user can select one of the items and press ENTER or can double-click it.

Allowing the user to select more than one item

You can allow the user to select more than one item from the List Box. The user can either:

♦ Select multiple items: Press CTRL when selecting the second and subsequent items or drag the mouse from the first item to the last.

♦ Make an extended selection: Select the first item and press SHIFT when selecting the last item, press CTRL when selecting the second and subsequent items, or drag the mouse from the first item to the last.

To allow more than one selection in a List Box:

1. Add a List Box form item to the form.
2. Double-click the List Box form item to display the Form List Box Options dialog box.
3. Click Options to display the Form Item Options dialog box.
4. Check the Extended Selection or Multiple Selection check boxes, as appropriate.
5. Click OK on each dialog box to save and exit.
Using smart list search
A List Box can use smart list search. Smart list search allows the user to quickly find an entry in a list by typing the first few characters of the entry in the list form item. If you would like to enable or disable smart list search, see “Enabling or disabling smart list search” on page 160.

Adding list items
Using the Designer, there are two ways to add list items to a List Box:

♦ Enter items manually. To enter list items manually:
  1. Double-click the List Box form item to display the Form List Box Options dialog box.
  2. In the entry field under the Labels list box, type the name of the first list item.
  3. Click Add or press ENTER to add the list item to the Labels list box.
  4. Repeat steps 2 and 3 for each list item that you want to add.
  5. Click OK to save your work and close the dialog box.

♦ Use the Link facility to declare a data source. For more information about the Link facility, see “Declaring the source of data for a form item” on page 146.

Creating synchronized lists
You can create synchronized List Boxes, which are a collection of List Boxes whose selections mirror the others. For example, consider two synchronized List Boxes: A and B. When the user selects item 3 in list A or in list B, the system selects item 3 in the other list.

To create synchronized List Boxes:

1. Add two or more List Boxes to the form. Add list items to each List Box and place them next to each other on a form.

   You cannot select Extended Selection or Multiple Selection from the Format Item Options dialog box when you create synchronized lists.

2. Remove the vertical scroll bar from each List Box except the rightmost one. (Double-click the List Box, click Options, and check the No Vert. Scroll Bar check box.)
3. Select each of the List Boxes. You must select at least one of the List Boxes before you can go on to step 4.

4. Select **Tools > Synchronized lists** from the Designer menu.

The Synchronized List Groups dialog box displays:

![Synchronized List Groups dialog box](image)

5. Click **New Group**.

The ID of the new group displays in the ID list box.

6. Select the List Boxes to synchronize from the Available Lists list box and click **Move**.

   - **To delete an item from a group, select it and click Delete. To delete a group, select its ID and click Delete Group.**

7. Click **OK** to save your work or click **Cancel** to exit without saving.

8. Save and test the interface.

   - **When you select an item from one List Box, the corresponding items in the other List Boxes also are selected.**
ListView

A ListView is a display box containing a list of selectable items. Each row of this list can be divided into resizable, labeled columns.

Select an item from the list. To select an item, a user can perform one of the following:

♦ Do the following:
  1. Click one of the list items.
  2. Press ENTER.
♦ Double-click one of the list items.

ListView display modes. A user can choose to display a ListView in the following modes:

♦ Large Icons
♦ Small Icons
♦ List
♦ Report

Smalltalk coding required. You can only configure a few aspects of a ListView from the Designer. To configure the rest, and to populate the list, you must write Smalltalk code. This section explains how to do so.

As an example, a ListView-style form in the Windows interface. The right-hand pane of the Windows Explorer window is a ListView-like display from which you can select folders and files. (The columns in this pane are only visible when you view the pane in Detail mode—equivalent to ObjectStudio’s “Report” mode—by selecting the View > Detail menu option.)
Creating a ListView

To create a ListView, perform the following:

1. Open a form.
2. Add a new ListView form item to the form.
3. Double-click the ListView form item.
   The Form ListView Options dialog box appears.
4. Perform one of the following:
   - Accept the default value that ObjectStudio places in the Name field (either “listView1” or something similar).
   - Enter a new name into the Name field.
5. If desired, enter some text into the Text field.

ObjectStudio does not use the Text field.
6. If desired, customize the ListView form item. To start this process, click the Options button.

When you click the Options button, the Form Item Options dialog box appears. The following table describes a ListView’s relationship to several options that are located in the Miscellaneous area of the Form Item Options dialog box.

<table>
<thead>
<tr>
<th>Radio button</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow Drag</td>
<td>Does not apply to a ListView.</td>
</tr>
</tbody>
</table>
| Allow Label Edit   | If selected, a user will be able to change the heading labels in a ListView. He or she will then be able to commit the label change by pressing ENTER, at which point the #labelChanged:to: event will be triggered.  
**Note:** If you select this checkbox, you must write a handler for your Controller. You can use the Event Editor to do this. |
| Auto Arrange       | Does not apply to a ListView.                                                                                                                                                     |
| Double Click       | Does not apply to a ListView.  
**Note:** Whenever a user double-clicks on a ListView list item, the #doubleClicked: event is triggered. You must write a handler for your Controller. You can use the Event Editor to do this. |
| Multiple Selection | If selected, enables the user to select multiple ListView list items. He or she can do so in the following ways:  
♦ Using CTRL: The user must hold down CTRL while selecting the second and subsequent list items.  
♦ Using SHIFT: The user must select an initial list item in a range of adjacent items, then press SHIFT while selecting the last item.  
♦ Clicking and dragging: The user must drag the mouse pointer from the first of several adjacent list items to the last. |
<table>
<thead>
<tr>
<th><strong>Radio button</strong></th>
<th><strong>Explanation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Icon</td>
<td>If selected, space for an icon will be reserved in the left-most ListView column.</td>
</tr>
<tr>
<td>Virtual List View</td>
<td>If selected, enables a special mode that more quickly displays large lists in the ListView. This special mode uses a different set of Smalltalk classes that allow ListView drawing at an earlier stage (as soon as all the visible items are computed). You can either populate a Virtual ListView statically or dynamically:</td>
</tr>
<tr>
<td></td>
<td>♦ <strong>Statically:</strong> If you would like to populate a Virtual ListView statically, from a value model, enter the following line into the openInitialization method that you must create in the controller containing the Virtual ListView:</td>
</tr>
<tr>
<td></td>
<td>listview1 callback: false.</td>
</tr>
<tr>
<td></td>
<td>This line of code sets the callback mode to “false,” which determines that the Virtual ListView will be statically populated. After adding this line of code, you can treat this Virtual ListView as if it were a normal ListView.</td>
</tr>
<tr>
<td></td>
<td>♦ <strong>Dynamically:</strong> If you would like to populate a Virtual ListView dynamically, from the Controller, enter the following into the openInitialization method that you must create in the controller containing the Virtual ListView:</td>
</tr>
<tr>
<td></td>
<td>listview1 callback: true</td>
</tr>
<tr>
<td></td>
<td>This line of code sets the callback mode to “true,” which determines that the Virtual ListView will be dynamically populated. After adding this line of code, you must create the getSizeForListView:, getobjectForListView:at:, getIconForListView:at: methods.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> For more information on populating a Virtual List View dynamically, and for code samples, see <em>Dynamically populating a ListView form item</em>.</td>
</tr>
</tbody>
</table>

For descriptions of the other options available in the Form Item Options dialog box, see “Setting form item attributes” on page 152.
7. If you would like to edit the font and colors used by the ListView, click the **Format** button on the Form ListView Options dialog box.

8. Click the **Methods** button.
   
   The Method Assignment dialog box appears.

9. As necessary, assign methods to the ListView.

10. On the Form ListView Options dialog box, click **OK**.

11. Save the form.

**Using smart list search**

A ListView can use smart list search. Smart list search allows the user to quickly find an entry in a list by typing the first few characters of the entry in the list form item. If you would like to enable or disable smart list search, see “Enabling or disabling smart list search” on page 160.

**Populating a ListView**

You can populate a ListView in either of the following ways:

♦ **Statically**

♦ **Dynamically**

**Statically populating a ListView.** To create the openInitialization method that will statically populate the ListView, perform the following:

1. Select the **Tools** > **Methods** menu option.
   
   The **Method Editor** appears.

2. Click the **New** button.

3. In the Source edit area, create a method that will populate the ListView statically.

   **Sample method:** For a sample openInitialization method that will statically populate the ListView, see the note on the following page.

4. Click the **Save** button.

5. Click the **Close** button.

6. Save the form.

7. Test the form.
Sample openInitialization for statically populating a ListView with a list of all currently open Forms. See the following:

```smalltalk
definition openInitialization
    | widths labels objects values |
    widths := \{ 14 12 \}. 
    labels := \{ 'Window Title' 'Controller' \}. 
    objects := ActiveForms asArray reversed. 
    values := objects collect: \[ :form | 
        Array with: form title with: form controller name \] . 
    listView1 putObjects: objects. 
    listView1 putValues: values. 
    listView1 widths: widths. 
    listView1 putLabels: labels. 
    listView1 setViewType: #Report. 
    listView1 redrawControllerFromValueModel.
```

**Note:** In this sample method, ActiveForms is a system Global Variable that holds a Collection of currently active Forms.
**Dynamically populating a ListView:** A dynamically-populated ListView pulls its contents from the Controller as it needs them.

A ListView form item can only be dynamically populated if you have defined it as a Virtual List View; to do this, select the Virtual List View checkbox on the Form Item Options dialog box. You must then define an openInitialization method in the controller that contains the Virtual ListView, as well as three more methods that ListView will use to pull its contents from your Controller. These three methods are:

- getSizeForListView:
- getObjectForListView:at:
- getIconForListView:at:

To create each of the four methods that will dynamically populate the ListView, perform the following:

1. Select the **Tools > Methods** menu option.
   
   The **Method Editor** appears.

2. Click the **New** button.

3. In the **Source** edit area, create the first of the methods that will populate the ListView dynamically.

**Sample methods:** See the following four notes for sample methods that dynamically populate a ListView with a list of all currently open Forms.

4. Click the **Save** button.

5. Click the **Close** button.

6. For each additional method that you need to create, return to step 2.

7. Save the form.

8. Test the form.
openInitialization method for dynamically populating the ListView with a list of all of the currently open Forms. The openInitialization sample method is below:

```plaintext
openInitialization
 | widths labels objects values |
forms := OrderedCollection new.
widths := { 14 12 }.
labels := { 'Window Title' 'Controller' }.
listView1 callback: true.
listView1 setViewType: #Report.
listView1 widths: widths.
listView1 putLabels: labels.
listView1 setViewType: #Report.
listView1 redrawControllerFromValueModel.
```

Note: The callback: method tells the List View that it will be dynamically populated.

getSizeForListView: method for dynamically populating the ListView with a list of all of the currently open Forms. The getSizeForListView: sample method is shown below:

```plaintext
getSizeForListView: cItem
^ ActiveForms size.
```

getObjectForListView:at: method for dynamically populating the ListView with a list of all of the currently open Forms. The getObjectForListView:at: sample method is below:

```plaintext
getObjectForListView: cItem at: index
 | form |
form := ActiveForms at: index.
^ Array with: form title with: form controller name
```

getIconForListView:at: method for dynamically populating the ListView with a list of all of the currently open Forms. The getIconForListView:at: sample method is below:

```plaintext
getIconForListView: cItem at: index
 | form |
form := ActiveForms at: index.
^ form icon isNil ifTrue: [ #None] ifFalse: [form icon].
```
Programming with a ListView

This section describes a number of programming tasks you can perform with ListViews:

♦ Setting the view type
♦ Setting extended styles using extended-style-specific methods
♦ Setting extended styles using a generic method with different arguments
♦ Using checkboxes in a ListView
♦ Using events generated by a ListView
♦ Programmatically scrolling a ListView
♦ Programmatically initiating a label change in a ListView
♦ Assigning a pop-up menu to a ListView
Setting the view type. To set the view type, use the `#setViewType:` method. For example, to set the view type to #Report, enter the following:

```small
listView1 #setViewType: #Report.
```

The following are possible view types:

- **Large icons.** Use the following argument for the `setViewType` method:
  ```small
  #Icon
  ```

- **Small icons.** Use the following argument for the `setViewType` method:
  ```small
  #SmallIcon
  ```

- **List (A list without details).** Use the following argument for the `setViewType` method:
  ```small
  #List
  ```

- **Report (A list with details).** Use the following argument for the `setViewType` method:
  ```small
  #Report
  ```

Setting extended styles using extended-style-specific methods. ListViews explicitly supports some extended styles, which only take effect when the ListView is using the #Report view type. You can explicitly set the following extended styles:

- **Full-row select.** To activate this extended style, use the following code:
  ```small
  listView1 setFullRowSelectTo: true.
  ```

- **Grid lines.** To activate this extended style, use the following code:
  ```small
  listView1 setGridLinesTo: true.
  ```

- **Checkboxes.** To activate this extended style, use the following code:
  ```small
  listView1 setCheckBoxesTo: true.
Setting extended styles using a generic method with different arguments.

ListViews support a number of extended styles by allowing the use of various arguments to the following generic method (in the following method sample, replace `#extended_style_argument` with one of the arguments in the following list):

```small
listView1 formItem setExtendedStyle: #extended_style_argument to: true.
```

ListViews can support the following ListView extended styles under the above generic method:

- **Checkboxes.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_CHECKBOXES`

- **Drag-and-drop the header.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_HEADERDRAGDROP`

- **Flat.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_FLATSB`

- **Full-row select.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_FULLROWSELECT`

- **Grid lines.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_GRIDLINES`

- **One-click activate.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_ONECLICKACTIVATE`

- **Track select.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_TRACKSELECT`

- **Underline cold.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_UNDERLINECOLD`

- **Underline hot.** Use the following argument with the `setExtendedStyle:to:` method:

  `#LVS_EX_UNDERLINEHOT`
Using checkboxes in a ListView. When you use the Report (that is, a list with details) view type with a ListView, the left-most column can display a checkbox in each row. To make it do so, enable the Checkbox extended style either explicitly or with a generic method:

- **Enabling checkboxes in a ListView.** You can either use an explicit or generic method:
  - Explicit method—Use the following code:
    ```plaintext
    listView1 setCheckBoxesTo: true.
    ```
  - Generic method—Use the following code:
    ```plaintext
    listView1 formItem setExtendedStyle: #LVS_EX_CHECKBOXES to: true.
    ```

- **Programmatically selecting or deselecting checkboxes in a ListView.** Use either of the following methods:
  - `setCheckState:` method—See the following code sample:
    ```plaintext
    listView1 setCheckState: index to: aBoolean.
    ```
  - `setCheckStates` method—See the following code sample:
    ```plaintext
    listView1 setCheckStates: anArrayOfBooleans.
    ```

- **Extracting the values of checkboxes in a ListView.** Use either of the following methods:
  - `getCheckState` method—See the following code sample:
    ```plaintext
    listView1 getCheckState: index.
    ```
  - `getCheckStates` method—See the following code sample:
    ```plaintext
    listView1 getCheckStates.
    ```
Using events generated by a ListView. Each ListView generates events as a user interacts with it.

- Handling events. If your application needs to react to these events, you must use the Event Editor to write handlers for your Controller.

- Using a ListView’s supported events. These events include the following:
  - #columnClicked: - Consider the following:
    - When it is triggered - A user clicks the title button of a ListView’s column.
    - Condition for triggering - ListView must have a #Report view type.
    - Normal response to triggering - Your application sorts the ListView.
  - #labelChanged:to: - Consider the following:
    - When it is triggered - A user commits the change of a label of the selected ListView item.
    - Condition for triggering - ListView has been defined with “Allow Label Edit” selected.
  - #beginDragHeader: - Consider the following:
    - When it is triggered - A user starts dragging a column header.
    - Conditions for triggering - The following:
      - The ListView has a #Report view type.
      - The #LVS_EX_HEADERDRAGDROP extended style has been enabled for the ListView.
- #endDragHeader: - Consider the following:
  - When it is triggered - A user stops dragging a column header.
  - Condition for triggering - The #LVS_EX_HEADERDRAGDROP extended style has been enabled for the ListView.

- #clicked:value: - Consider the following:
  - When it is triggered - A user selects or deselects a checkbox in the ListView.
  - Conditions for triggering - The following:
    - The ListView has a #Report view type.
    - The #LVS_EX_CHECKBOXES extended style has been enabled for the ListView.

- #checked: - Consider the following:
  - When it is triggered - A user selects a checkbox in the ListView.
  - Conditions for triggering - The following:
    - The ListView has a #Report view type.
    - The #LVS_EX_CHECKBOXES extended style has been enabled for the ListView.

- #unchecked: - Consider the following:
  - When it is triggered - A user deselects a checkbox in the ListView.
  - Conditions for triggering - The following:
    - The ListView has a #Report view type.
    - The #LVS_EX_CHECKBOXES extended style has been enabled for the ListView.
**Programmatically scrolling a ListView.** You can programmatically scroll the List View using either of the following methods:

- **setTopTo: method.** See the following code sample:
  ```smalltalk
  listView1 setTopTo: index.
  ```

- **ensureVisible:partialOk: method.** See the following code sample:
  ```smalltalk
  listView1 formItem ensureVisible: index partialOk: aBoolean.
  ```

**Programmatically initiating a label change in a ListView.** You can programmatically initiate a label change for a programmatically-selected ListView label or for the currently-selected ListView label:

- **Programmatically-selected label.** See the following code sample:
  ```smalltalk
  listView1 startEditLabelAt: index.
  ```

- **Currently-selected (currently-selected) label.** See the following code sample:
  ```smalltalk
  listView1 startEditLabel.
  ```

**Assigning a pop-up menu to a ListView.** You can assign a pop-up menu to a ListView either statically or dynamically:

- **Statically.** Use the following method:
  ```smalltalk
  listView1 setPopupMenuTo: (cItemDict at: #Menu).
  ```

- **Dynamically.** To dynamically assign a pop-up menu, handle the #wmButton2DownAt: event.

  If a pop-up menu is statically assigned to a ListView, the #wmButton2DownAt: event will not be triggered. Therefore, if you wish to dynamically assign a pop-up menu to a ListView, make sure that no pop-up menu is statically assigned to the ListView. To make sure of this, use the following code:

  ```smalltalk
  listView1 setPopupMenuTo: nil.
  ```
Multi Line Entry Field

A Multi Line Entry Field is an Entry Field that displays more than one line of text or accepts more than one line of text.

For more information on Entry Fields, see “Entry Field” on page 187.

Notebook

A Notebook consists of a set of related tabbed pages, each of which allows you to change settings related to one aspect of the application. Notebook options include specifying the size, shape, and position of tabs, and the page orientation.

Notebook limitations. Each Notebook page is treated as a separate window. When the user opens a Notebook, the system opens all Notebook pages at one time. So, keep these suggestions in mind as you design a Notebook:

♦ Limit the number of pages in a Notebook. When a Notebook contains more than 10 pages, system performance starts to degrade significantly.
♦ Your operating system may impose a limit on the number of items that can be open at one time. If a Notebook contains more pages than can be open at one time, an error can result.

Creating a Notebook

To create a Notebook:

1. Add a Notebook form item to the form.
2. Double-click the Notebook form item to open the Form Notebook Options dialog box.
   Do not double-click the Notebook, but double-click near the edge of the form item.
The following table describes the pages in the Notebook:

<table>
<thead>
<tr>
<th>Page</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions</td>
<td>The dimensions of the major tabs, the minor tabs (subdivisions of major tabs), and the page buttons (arrows that allow you to view the previous or next page).</td>
</tr>
<tr>
<td>Style</td>
<td>The shape of tabs, the location of back pages, and whether the presentation of the Notebook should display to be spiral bound.</td>
</tr>
<tr>
<td>Locations</td>
<td>The side of the Notebook along which the tabs should display and where status information should display.</td>
</tr>
<tr>
<td>Link</td>
<td>The Notebook’s data source. For more information, see “Declaring the source of data for a form item” on page 146.</td>
</tr>
</tbody>
</table>
Options button. Click Options to change the Miscellaneous and Keys Form Item Options.

Pages button. Click Pages to add Regular (untabbed), Major (tabbed), or Minor (tabbed, subdivisions of Major) pages to the Notebook. Then:

♦ To add a page, type the name of the page in the Page Name entry field and select the type of page you want it to be. Click Add.

♦ To delete a page, select it in the large list box and click Delete Page.

Adding items to a Notebook page
To add items to a Notebook page:
1. Select Form > Page name from the Designer menu.
2. Add form items to the page.
3. Repeat steps 1 and 2 for each page in the Notebook.

There are many methods that affect Notebooks. For more information about these methods, use the Class Browser to look up methods for the classes NoteBookCtrl, Page, and FormNoteBook.
Number Field

A Number Field is an Entry Field that allows you to format the display of numbers and verify the entry of numeric values. For more information, see “Entry Field” on page 187.

When the user enters text into the field and sets focus to another form item, the system determines whether the text is a number. If it is a number, the system reformat the text according to the format you associate with the field. If it is not a number, the system beeps.

Make certain that the Windows Default sound is set to a value other than None, or no beep will sound. Alter this setting through the Sounds control panel.

Shortcuts

Using the m shortcut. When a user enters the letter m into a Number Field, the system replaces it with three zeros.

Defining a new shortcut. You can create a new Number Field shortcut, similar in function to the m shortcut described above. You can define the character that will be used for your new shortcut, as well as the behavior that the character will represent.

For every Windows character event (such as a keypress) that affects an ObjectStudio form item, a characterEvent: method in an appropriate subclass of ControllerItem is synchronously called. A characterEvent: method can perform special actions when a certain character is entered.

For a Number Field, the appropriate subclass of ControllerItem is NumberCtrl. Therefore, the NumberCtrl>>characterEvent: method must be modified in order to create a new Number Field shortcut.

Make certain you cause the NumberCtrl>>characterEvent: method to return a result of true when your shortcut character is entered. When a result of true is returned, primitive processing for the shortcut character is skipped. This keeps the system from beeping if the Beep on errors option has been selected for the Number Field.
As an example, upper- or lower-case “k” can be defined as a shortcut key in a Number Field by the addition of the following sample code to the NumberCtrl>>characterEvent: method:

```
super characterEvent: aChar.
(aChar = $k) | (aChar = $K) ifTrue: [
    self put: self value * 1000.
    ^ true.
].
```

After this sample code has been added, entering a number accompanied by an upper- or lower-case "k" into a Number Field causes that number to be multiplied by 1000. For example, if the user enters 100k, the Number Field displays 100000.

**put: does not validate**

If you use the put: method to place data in a Number Field, the system does not format or validate the data. Therefore, if you add data to a Number Field programmatically, ensure that the data is valid and formatted correctly.

To learn more about specifying the format of a Number Field, see “Changing the format of a form item” on page 139. To learn more about how the system validates data and about how to change the system’s default validation behavior, see “Validating data” on page 357.
User input testing

During the user input process, you can test a new value entered into a Number Field. When a new value is entered into a Number Field, that value will be held in a temporary variable. A method (either `userInputVerificationFor:` or `userInputVerificationFor:value:`), which you will have customized, will test the entered value. If the method returns "true," the value in the temporary variable will be moved into the Number Field. If not, the Number Field will remain as it was before the new value was entered.

You can create and customize two different methods that will test Number Fields:

♦ `userInputVerificationFor: aValue`. Create and customize this method in class `NumberCtrl` if you would like every Number Field in your application to perform the same test when a new value is entered. The argument is the value that was entered into the Number Field.

♦ `userInputVerificationFor: ctrlItem value: aValue`. Create and customize this method in the controller containing the Number Field or Fields that you would like to have test entered values. Use this method if you would like only certain Number Fields to test entered values, or if you would like different Number Fields to use different tests. The first argument is the controller item. The second argument is the value that was entered into the Number Field.
Example. If you would like to create a Number Field that accepts only the values ranging from -99.99 to 99.99, inclusive:

1. Using the Designer, create a new Number Field.
2. Format the Number Field so that it has a precision of 2.
3. Using the Method Editor, create and save the following `userInputVerificationFor:value:` method in the controller containing the Number Field you have just created:

   ``` Smalltalk
   userInputVerificationFor: ctrlItem value: aValue
   | aNumber |
   (ctrlItem isKindOf: NumberCtrl) & (aValue notNil)
   ifTrue: [
     aNumber := aValue getValue.
     aNumber notNil ifTrue: [
       ^ aNumber abs < 100.
     ].
   ].
   ^ true.
   ```

   This method has been written to test all Number Fields in the controller, not only the Number Field you have just created.
OLE Custom Control

OLE Custom Controls are reusable pieces of software that you can plug into an application. One common use of an OLE Custom Control is as a supplemental form item that is available to the application.

For more information about creating OLE Custom Controls, refer to the ObjectStudio OLE User's Guide, P40-3805.

You can use the Automation Function Helper to help you create a method for an OLE Custom Control. For more information, see “Automation Function Helper” on page 65.

OLE Item

The OLE Item form item creates an OLE site that holds an embedded or linked object on the container (the form on which you place the OLE Item).

For more information about creating OLE Items, refer to the ObjectStudio OLE User's Guide, P40-3805.
Progress Bar

A Progress Bar indicates how much progress has been made for a given task. For example, installation programs frequently use this type of form item to indicate what percentage of the installation is complete.

Creating a Progress Bar

To create a Progress Bar:

1. Add a Progress Bar form item to the form.
2. Double-click the Progress Bar form item to open the Progress Bar Options dialog box, as shown here:
The following table describes the fields on the dialog box:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Smalltalk name for the Progress Bar; for example, progressBar1.</td>
</tr>
<tr>
<td>Lower Range</td>
<td>The lower value of the range of values that the Progress Bar displays. For example, for a Progress Bar that displays progress from 0 to 10, enter 0 here.</td>
</tr>
<tr>
<td>Upper Range</td>
<td>The upper value of the range of values that the Progress Bar displays. For example, for a Progress Bar that displays progress from 0 to 10, enter 10 here.</td>
</tr>
<tr>
<td>Step</td>
<td>The increments to display between the lower range and upper range. For example, for a Progress Bar that ranges from 0 to 10 in increments of 2, enter 2 here.</td>
</tr>
</tbody>
</table>

**Programmatic interface**

The setRangeLower:upper: method sets the upper and lower range of a Progress Bar. For example, the following code sets the lower range to 1 and the upper range to 10 for a Progress Bar named cprogbar:

```smalltalk
cprogbar setRangeLower: 1 upper: 10.
```

The step: method sets the number of units that the cprogbar advances each time. For example, the following method sets the number of units that the cprogbar advances each time to 2:

```smalltalk
cprogbar step: 2.
```

The stepIt method advances the Progress Bar. For example, the following code advances the cprogbar:

```smalltalk
cprogbar stepIt.
```

**Limitations**

Note that the width of each Progress Bar tick is static. If you specify a range that contains more ticks than the Progress Bar can hold, the ticks can wrap. Therefore, you may need to change the size of the Progress Bar to accommodate the number of ticks you specify.
Radio Buttons

A set of Radio Buttons presents the user with an exclusive choice. For example, a set of Radio Buttons can allow the user to select a size: Small, Medium, or Large. To allow multiple selections, use a Check List (see “Check List” on page 173).

Display-only Radio Buttons

You can create a set of Radio Buttons that is read only. For example, you can display the result of a choice made elsewhere in the application but not allow the user to change the choice. To do so, check the Not Editable and Not Selectable check boxes on the Form Item Options dialog box.

Turning off default selection

You can set the Radio Buttons so that when they are first displayed; there is no default selection. To do so, send the setSelectionTo: message with argument 0 (zero).
Rectangle

A Rectangle is a scrolling box that allows you to display tables of data. Each piece of information displays in an individual cell. Compare the Rectangle to a Tabular List Box, which displays data one row at a time. A Rectangle is field-oriented, and a Tabular List Box is row-oriented.

The cells in a Rectangle are numbered, starting with cell (0,0). Row 0 and Column 0 are reserved for labels.

Displaying data

To add data to a Rectangle, use the Link facility to declare a data source. If you use the Link facility, the class generated is FormTable, which is a subclass of Rectangle. For more information about the Link facility, see “Declaring the source of data for a form item” on page 146.

Alternately, you can add data to a Rectangle by subclassing methods in the RectCtrl and FormRect classes (for example, getColLabelsFrom:to:, getRectValueFor:at:, and getRowLabelsFrom:to:). Use the Class Browser to see which methods are available.

Using a Rectangle to display a table

This example shows how to use a Rectangle to display the Employee table, as shown here:
Form items. To add the form items:

1. Select FormItem > New Item from the Designer menu to open the New Item dialog box.

2. Select the Rectangle form item and then:
   A. Type the following in the Name entry field on the New Item dialog box:
      table
   B. Press ENTER to add the Rectangle form item to the main form.
   C. Move the form item to the upper-left corner of the form and resize.

3. Select the Static Text form item and then:
   A. Type the following in the Text entry field on the New Item dialog box:
      Selected Text:
   B. Press ENTER to add the Static Text form item to the main form.
   C. Move the form item below the Rectangle form item.

4. Select the Entry Field form item and then:
   A. Type the following in the Name entry field on the New Item dialog box:
      display
   B. Press ENTER to add the Entry Field form item to the main form.
   C. Move the form item next to the Static Text form item.

Link the Rectangle to the Employee table. To link the Rectangle to the Employee table:

1. Double-click the Rectangle item to open the Form Rectangle Options dialog box.

2. Click the Link button to open the Link Hookups dialog box.

3. Select Table in the left-most list box, and then Employee in the list box to the right of the left-most one.

4. Click OK to close the Link Hookups dialog box.

5. Click OK to close the Form Rectangle Options dialog box.
**Retrieve the selected value.** To retrieve the selected value and display it in the Entry Field:

1. Select **Tools > Methods** from the Designer menu to display the **Method Editor**.
2. Create the following method:
   - A. Click **New** on the Method Editor.
   - B. In the **Source** edit area, type:
     ```
     select
display put: (table getValue).
     ```
   - C. Click **Save** to save the method.
3. Click **Close** to close the Method Editor.

**Add a method.** To add a method and select options for the Rectangle form item:

1. Double-click the Rectangle form item to open the Form Rectangle Options dialog box.
2. Click **Options** to open the Form Item options dialog box, then:
   - A. Select the **Horz. Resizeable** and **Vert. Resizeable** check boxes.
   - B. Click **OK** to close the Form Item Options dialog box.
3. Click **Methods** to open the Method Assignment dialog box, then:
   - A. Select the **select** method from the **Available Methods** list box.
   - B. Click **Copy** to add the method to the **Assigned Methods** list box.
   - C. Click **OK**.
4. Click **OK** to close the Form Rectangle Options dialog box.
Select options. To select options for the Static Text and Entry Field form items:

1. Double-click the Static Text form item to open the Form Static Text Options dialog box, then:
   A. Click Options.
   B. Select the Vert. Moveable check box.
   C. Click OK to close the Form Item Options dialog box.
   D. Click OK to close the Form Static Text Options dialog box.

2. Double-click the Entry Field form item to open the Form Entry Field Options dialog box, then:
   A. Click Options.
   B. Select the Not Selectable and the Vert. Moveable check boxes.
   C. Click OK to close the Form Item Options dialog box.
   D. Click OK to close the Form Entry Field Options dialog box.

Save and test the example. To save and test the Rectangle example:

1. Select File > Save as from the Designer menu.
2. Select a folder in which to save the file.
3. Type a file name in the File name entry field. For example, type tableex.cls.
4. Click Save to save the file and close the Save As dialog box.
5. Click Test interface on the Designer toolbar.
6. Select a field in the table.
7. Click the field and note that the selected text displays in the Entry Field.
Scatter Graph

A Scatter Graph displays pairs of related numbers on a graph with X- and Y-axes.

Mouse options

The following table describes the options on the Form Item Options dialog box that are related to the mouse:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning when checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Mouse</td>
<td>Allows the user to select an item from the Scatter Graph.</td>
</tr>
<tr>
<td>Track Mouse</td>
<td>Allows the system to track the mouse as it moves over the Scatter Graph.</td>
</tr>
</tbody>
</table>
Creating a Scatter Graph
To create a Scatter Graph that compares the age and salary of an employee:

1. Add a Scatter Graph form item to the form and resize it to double its original height and width.
2. Double-click the Scatter Graph to display the Form Scatter Graph Options dialog box.
3. Define the data along the X-axis:
   A. Select the Insert X radio button.
   B. Click Link.
   C. On the Link Hookups dialog box, from left to right, select Table, Employee, and Salary.
   D. Click OK.
4. Define the data along the Y-axis:
   A. Select the Insert Y radio button.
   B. Click Link.
   C. On the Link Hookups dialog box, from left to right, select Table, Employee, and Age.
   D. Click OK.
5. Assign the color of the points on the Scatter Graph:
   A. Select an expression from the Expression List list box.
   B. Select a color from the Color drop-down list box.
6. Assign the shape of the points on the Scatter Graph:
   A. Select an expression from the Expression List list box.
   B. Select a shape from the Markers drop-down list box.
7. Click OK to exit the Form Scatter Graph Options dialog box.
8. Save and test the interface.
A Scatter Graph comparing age and salary is displayed.

Slider
A Slider allows the user to select a value from a range of discrete numeric values. You can use the Form Slider Options dialog box to specify start, stop, increment, and skip values.

The default Slider is vertical. However, you can make the Slider horizontal (click Options on the Form Slider Options dialog box). You can link the Slider to a data source.
Spin Button
A Spin Button presents a closed loop of items from which the user can make a choice.

Creating a Spin Button
To create a Spin Button that allows the user to choose one day of the week:
1. Add a Spin Button form item to the form.
2. Double-click the Spin Button to display the Form Spin Button Options dialog box.
3. For each day of the week:
   A. Place the cursor in the entry field that is between the large list box and the OK button.
   B. Enter each day of the week, as shown here:
4. Click **OK** to save your work and exit the Form Spin Button Options dialog box.

5. Save and test the interface.

When you click one of the Spin Button arrows, the Spin Button displays the next item in the list.

**Static Text**

A **Static Text** form item serves as a text label for items that you place on forms. The user cannot change static text, and a Static Text form item cannot respond to key presses.

You can add a mnemonic to a Static Text form item. For more information about adding mnemonics, see “Customizing form items” on page 137.

**Tabular List Box**

A **Tabular List Box** is a scrolling List Box that presents tabular data, such as data from a database table. Each item in the Tabular List Box represents one row of data, one row at a time. When you click a column header, the Tabular List Box is sorted by the data in that column, toggling between an ascending and descending sort; this behavior is similar to that of a ListView. Compare the Tabular List Box to a Rectangle, which presents each piece of data in its own cell.

There are two ways to add data to a Tabular List Box:

- **Use the Link facility to declare a data source**: For more information, see “Declaring the source of data for a form item” on page 146.

- **Use methods in the TabListCtrl class**: Use the Class Browser to see the list of available methods.

**Split list**

If you check the Split list check box on the Form Tabular List Box Options dialog box, you can split the Tabular List Box into two vertical halves. Once split, the two halves scroll horizontally and independently of each other.
Grid lines

You can display horizontal and vertical grid lines between data cells on a Tabular List Box. Below on the left is a graphic of a Tabular List Box without grid lines; below on the right is a graphic of a Tabular List Box with grid lines:

Using the Designer. There is now a Lined checkbox on a Form Item Options dialog box for a Tabular List Box. Check this box to display grid lines; uncheck this box to hide grid lines.

Using code. You can execute code to dynamically display or hide grid lines on a Tabular List Box:

- **Displaying grid lines.** Execute code similar to the following sample:

  ```ruby
  | fItem options | "on"
  fItem := tabularListBox1 formItem.
  options := fItem options.
  (options hasElement: #Lined ) ifFalse:
  [ options add: #Lined.
  fItem setOptionsTo: options; redrawWindow.
  ].
  
  - **Hiding grid lines.** Execute code similar to the following sample:

  ```ruby
  | fItem options | "off"
  fItem := tabularListBox1 formItem.
  options := fItem options.
  (options hasElement: #Lined ) ifTrue: [ 
  options remove: #Lined.
  fItem setOptionsTo: options; redrawWindow.
  ].
  ```
Data classes and conditional icons

Workplace Tabular List Boxes, a subclass of Tabular List Boxes, support the use of data classes and conditional icons.

The ObjectStudio Workplace Demo EmployeeInfo controller implements a data class, EmployeeRecord, and conditional icons. Examine this file to learn more about the use of data classes and conditional icons.

To create a new Data Class entry:

1. Double-click the Tabular List Box form item to open the Form Tabular List Box Options dialog box.

2. Click New.

The New Data Class Entry dialog box displays, as shown here:
3. Type the required information in the Class Specification entry fields:
   A. Application - Name of the controller to which the class is assigned.
   B. Database Name - Name of the database to use with this class.
   C. Table Name - Name of the table in the database to use.
   D. Default Name - Default name of objects created with this List Box and a template.

4. Type the required information in the Object Specification entry fields:
   A. Column Names - Array of table column names used for this class.
   B. Editor - Name of the editor to use to make changes to the table.
   C. Name - Field in the table that specifies the name of a workplace object.
   D. Is Empty - Expression that returns if the selected record is empty.
   E. Unique Id - A unique SQL WHERE clause that points to a specific record.
   F. Unique Col's - Display column names that include unique values.

5. Click Icon Def to open the Icon Definition dialog box and define the default icon and conditional icons. Enter conditions as arithmetic expressions (for example, salary<30000).

6. Click OK to save changes and close the New Data Class Entry dialog box.

7. Click OK to close the Form Tabular List Box Options dialog box.

**Smart list search**

A Tabular List Box always uses smart list search, which enables a user to quickly locate a list entry by typing the first few characters of the entry in the list form item. Smart list search is always enabled for a Tabular List Box; although you can disable smart list search for other kinds of list form items, you cannot disable it for Tabular List Boxes. For more information about smart list search, see “Enabling or disabling smart list search” on page 160.
Sorting by clicking a column header

By default, clicking a column header causes ObjectStudio to sort by the data in that column, toggling between ascending and descending sort and then triggering the #columnNameClicked: event.

Disabling or customizing column header click sorting

If the columnNameClicked: event is hooked, the Tabular List Box is not automatically sorted when a user clicks a column header. One consequence of this is that Tabular List Boxes can be made nonsortable. To accomplish this, hook an empty method to the columnNameClicked: event on a Tabular List Box.

The sorting behavior in Tabular List Boxes can be customized. To accomplish this, add the code that performs your custom sorting to the method that you have hooked to the columnNameClicked: event. For example, you can add code that sorts the Tabular List Box data using more than one column.

You can preserve the default Tabular List Box sorting behavior even if the columnNameClicked: method is hooked. To accomplish this (for example, for backwards compatibility), add the code that performs the default sorting to the method that you have hooked to the columnNameClicked: event. For example, add the following code to this method:

```
tabList1 reSortByColumnName: aName
```

Data sometimes converted to Strings before sort

If the data in the column of the Tabular List Box being used for the sort understands the message #<, then ObjectStudio will send #< to the column value directly in order to determine the sort order. Otherwise, data is converted to Strings before it is sorted.

Event that detects a data-row click

There is an event that fires when the user clicks a data row on the Tabular List Box:

```
clickedOnColumn:row:
```

The first parameter, columnName, is derived from the layout of the clicked row. The second parameter, rowNumber, is the number of the clicked row.
Text collector

A Text Collector is a scrolling box that contains lines of text. Although the ObjectStudio System Transcript window is not implemented as a Text Collector, it is an example of a Text Collector.

In the following example, you create an interface with a Text Collector and two Buttons. When you click Add Text, the system adds a line of text to the Text Collector. When you click Clear, the system clears the Text Collector.

To create a Text Collector:

1. Add the following items to the form:
   A. A Text Collector
   B. A Button with text: Add Text
   C. A Button with text: Clear

2. Select Tools > Methods from the Designer menu to open the Method Editor.

3. Create the following method:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:

   ```plaintext
   writeToTC
   textCollector1
   nextPutAll: 'This is a test';
   cr;
   refresh.
   ```
   C. Click Save to save the method.
4. Create the following method:
   A. Click New on the Method Editor.
   B. In the Source edit area, type:
      
      clearTC
textCollector1 clear.
   C. Click Save to save the method.
   D. Click Close to close the Method Editor.

5. Use the Method Assignment dialog box to link the writeToTC method to the Add Text button.

6. Use the Method Assignment dialog box to link the clearTC method to the Clear button.

7. Save and test the interface.
   When you click Add Text, the system displays the text “This is a test” in the Text Collector, as shown here:

   ![Image of text collector with “This is a test”](image)

   When you click Clear, the system clears the Text Collector.
Text Editor

A Text Editor item provides a fully functional editor for changing Smalltalk class files and other text files. It includes a regular search facility and the ability to load and execute Smalltalk code. It also is extensible.

For more information about how to use the Text Editor, refer to the ObjectStudio User’s Guide, P40-3807.

Topic Box

A Topic Box is a labeled box that surrounds other form items. Use it to visually group a set of logically related form items.

Topic Box items include a static text label and a rectangular frame used to surround associated items on a form. You can apply text formatting attributes and change the typeface and color of the static text.

You can add a mnemonic to a Topic Box form item. For more information about adding mnemonics, see “Customizing form items” on page 137.
Tree View

A Tree View allows you to display hierarchical information in a logical manner. The user can expand a level of the hierarchy to view the next lower level. The user can collapse expanded levels.

The tree is the entire collection of items. Each item in the tree is a node. If a node has levels below it in the hierarchy, you can expand or collapse the node. If a node has no levels below it in the hierarchy, it is a leaf node.

Using a Tree View

Within the ObjectStudio development environment, the Select Controller Type dialog box contains a Tree View that allows you to view and select the type of controller on which to base an interface.

The following figure shows an expanded Tree View in a Windows Help file:

Other possible uses for a Tree View are to display an outline, display a Class Browser, or display a file directory structure.
Creating a Tree View

To create a Tree View:

1. Add a Tree View form item to the form.
2. Double-click the Tree View form item to display the Form Tree View Options dialog box, as shown here:

   ![Form Tree View Options dialog box]

   3. Type a new name in the Name entry field, or accept the default.
4. Type new values in the Indentation per Level, Icon Width, and Icon Height entry fields, or accept the defaults.

   The following table describes the settings on the Form Tree View Options dialog box:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indentation per Level</td>
<td>Number of characters to indent each new level.</td>
</tr>
<tr>
<td>Icon Width</td>
<td>Width of the icon in pixels.</td>
</tr>
<tr>
<td>Icon Height</td>
<td>Height of the icon in pixels.</td>
</tr>
</tbody>
</table>

5. Click Options to change the default Form Item Options (see “Changing Tree View display options” on page 236).
6. Click Format to change the font and colors of the Tree View.
7. Click Methods to display the Method Assignment dialog box, in order to assign methods.
8. Click OK to close the Form Tree View Options dialog box.
9. Save the interface.
Changing Tree View display options

Click **Options** on the Form Tree View Options dialog box to display the Form Item Options dialog box, as shown here:
The following table describes the options in the Miscellaneous topic box that are specific to a Tree View:

<table>
<thead>
<tr>
<th>Option</th>
<th>Allows you to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow Drag</td>
<td>Drag and drop a label to a different location in the Tree View.</td>
</tr>
<tr>
<td>Allow Label Edit</td>
<td>Change the text in the label.</td>
</tr>
<tr>
<td>Double Click</td>
<td>Double-click the icon or the text to expand or collapse the Tree View.</td>
</tr>
<tr>
<td>Dynamic Expand</td>
<td>Create nodes dynamically (for example, creating children nodes when the parent is expanded).</td>
</tr>
<tr>
<td>Reduced Tree</td>
<td>Display only the branches of the node that you choose to expand, if enabled. The default is to show the entire Tree View when you expand one node.</td>
</tr>
<tr>
<td>Show Dots</td>
<td>Display three periods before each node that is expandable, if enabled.</td>
</tr>
<tr>
<td>Show Icon</td>
<td>Enable or disable the display of icons.</td>
</tr>
</tbody>
</table>

For a description of the other options in the Miscellaneous topic box, see “Setting form item attributes” on page 152.

**Allow Drag.** The Allow Drag option allows the user to drag a label and drop it in another location within the Tree View only. When the drag and drop is done, the #drop:at: event is triggered. You must write a handler method for your controller; for example:

```ruby
onDrag: item1 to: item2
    treeView1 moveNodeFrom: item1 to: item2.
```

The moveNodeFrom:to: method updates the Tree View node list to reflect the move and updates the Tree View user interface.
Allow Label Edit. The Allow Label Edit option allows the user to change the text in the label. When the user presses ENTER to commit the changes, the \#labelChanged:to: event is triggered. You must write a handle for your controller; for example:

```plaintext
  onLabelChanged: item to: string
      treeView1 setTextAt: item to: string.
```

The `setTextAt:to:` method updates the label in the Tree View. Also, update the Tree View node, or the label can be lost.

Reduced Tree. The Reduced Tree option changes the way you can expand a Tree View. Consider the following hierarchy:

- Object
  - Boolean
  + Collection

If you do not select the Reduced Tree check box, the following Tree View displays when you expand the Boolean node:

- Object
  - Boolean
    + True
    + False
  - Collection
    + Bag
    + ByteCodeDecoder

If you select the Reduced Tree check box, the following Tree View displays when you expand the Boolean node:

- Object
  - Boolean
    + True
    + False
  + Collection
Displaying icons for a Tree View

Each node of the Tree View is represented by an icon that reflects the state of the node and by label text. By default, items that you can expand have a plus sign (+) icon next to them, and items that can collapse have a minus sign (−) icon next to them. You can change the default icons programmatically by setting the icon instance variable in the TreeViewNode class; for example:

```plaintext
openInitialization
| subNode root node |
subNode := IdentityDictionary new.
subNode at: #Object put: (IdentityDictionary new).
subNode at: #MyObject put: (IdentityDictionary new).
root at: #Root put: subNode.
node := TreeViewNode newDictionaries: root.
node deepDo: [:aNode|
  (aNode label == #Root) ifTrue: [aNode icon: (Bitmap getNamed: #File)].
  (aNode label == #Object) ifTrue: [aNode icon: (Bitmap getNamed: #DownArrow)].
  (aNode label == #MyObject) ifTrue: [aNode icon: (Bitmap getNamed: #CheckBoxes)].
].
treeView1 formItem imageSize: (16 @ 16).
treeView1 put: node.
```
Populating the Tree View

There are two ways you can populate the Tree View:

♦ Statically
♦ Dynamically

Statically. To create a Tree View statically, perform the following:

1. Select Tools > Methods to open the Method Editor.
2. Create the following method, which fills the Tree View with all classes to show the contents of the Class Browser:
   
   A. Click New on the Method Editor.
   
   B. In the Source edit area, enter the following:
   
   ```
   openInitialization
   | node |
   node := TVNode newDictionaries:
   (Object classNameTree at: #Collection).
   node deepSortUsing: [:node1 :node2 |
   node1 label > node2 label].
   treeView1 put: node.
   ^super openInitialization.
   ```
   
   Object classNameTree is a system method that returns the class hierarchy, which can then be used in a Tree View.

   C. Click Save to save the method.

3. Click Close to close the Method Editor.

4. Save and test the interface

The following figure shows the Tree View created by this method:

![Tree View Example]
Dynamically. You can put just the root of the tree into the Tree View and generate the branches when you expand the nodes.

To create a Tree View dynamically:

1. Select Tools > Methods to open the Method Editor.

2. Create the following method:
   
   A. Click New on the Method Editor.
   
   B. In the Source edit area, type:

   ```plaintext
   openInitialization
   | drive aDict root treeViewNode oldDir newDir | 
   drive := FileStream defaultDrive + DirSeparator.
   aDict := Dictionary new.
   aDict at: drive put: (Dictionary new).
   root := TreeViewNode newDictionaries: aDict.
   
   treeView1 put: root.
   
   treeView1 expandBlock: [aNode | 
   ^((FileStream subdirectoriesIn: (aNode value))
    removeAll: ['.','.'])
   collect: [:each |
   oldDir := aNode value asFileNameString.
   newDir := oldDir addFileComponent: each.
   treeViewNode := TreeViewNode new.
   treeViewNode label: each;
   level: (aNode level + 1);
   value: newDir;
   isExpanded: false;
   expandable: true;
   yourself.
   ].
   ].

   C. Click Save to save the method.

3. Click Close to close the Method Editor.

   The previous method displays directories and paths. Put the root in the Tree View and define an expandBlock so that the node can find its children when a request is made for them.
Value Set

A Value Set is a graphical array of items from which the user can select one item at a time. The New Item dialog box in the Designer is an example of a Value Set.

Creating a Value Set

To create the Value Set shown here:

1. Add a Value Set form item to the form.
2. Double-click the Value Set form item on the form to display the Form Value Set Options dialog box, as shown here:
3. Type the number of rows and columns to assign to the Value Set in the Rows and Columns entry fields. For this example, assign 2 Rows and 3 Columns.

The number of items in the Value Set equals the number of rows times the number of columns.

4. Add an item with a bitmap or icon file:
   A. From the Value type topic box, select the File radio button.
   B. In the entry field below the Labels list box, type the file name of a bitmap or icon file. For this example, type the full pathname to the ObjectStudio file icons\folclose.bmp. The default path is the following, with x.y replaced by the ObjectStudio version number:
      ```
      C:\Program Files\Cincom\ObjectStudio.x\ObjectStudio\icons\folclose.bmp
      ```
   C. Click Add or press ENTER.

5. Add an item with text:
   A. From the Value type topic box, select the Symbol radio button.
   B. In the entry field below the Labels list box, type the text that you want to display on the item. For this example, type Info.
   C. Click Add or press ENTER.

6. Add an item with a bitmap:
   A. From the Value type topic box, select the Bitmap radio button.
   B. In the entry field below the Labels list box, type the name of a bitmap. For this example, type LeftArrow.
   C. Click Add or press ENTER.
7. Add an item with an icon:
   A. From the **Value type** topic box, select the **Icon** radio button.
   B. In the entry field below the **Labels** list box, type the name of an icon. For this example, type Info.
   C. Click **Add** or press **Enter**.

8. Add an item that is a solid color:
   A. From the **Value type** topic box, select the **Color** radio button.
   B. In the entry field below the **Labels** list box, type the name of a color. For this example, type Black.
   C. Click **Add** or press **Enter**.

9. Click **OK**.
The Value Set with the items you created displays.

One item is blank because you only created five items for a six-item Value Set.

**Vertical border**

A vertical border is a vertical moveable border that enables a user to resize lists, Tree Views, text fields, or other items. For more information, see “**Moveable Borders**” on page 156.

**Workplace List Handler**

A Workplace List Handler is used for creating workplace applications. It acts as a container for other items.

To learn more about workplace applications, see “**5. Creating a workplace application**” on page 277.
Workplace Object Handler

A Workplace Object Handler is used to create workplace applications. It represents an object that stores data or performs an action.

To learn more about workplace applications, see “5. Creating a workplace application” on page 277.

Workplace Object Template

A Workplace Object Template is used for creating workplace applications. It represents a template for creating new workplace objects.

To learn more about workplace applications, see “5. Creating a workplace application” on page 277.
4. Working with interfaces and interface components

Interfaces and interface components

When you create a new user interface item using the Designer, you typically create a collection of form items that you can treat as one unit. When you create new items programmatically, you typically specialize an existing item.

Components of a UI item

Recall that a user interface item has two components:

♦ Form item: An instance of a subclass of FormItem. It manages interactions with the operating system.

♦ Controller item: An instance of a subclass of ControllerItem. It manages interactions with the controller.

The previous chapters showed you how to start working with the Designer by creating a new interface based on the Controller. This chapter shows how you can create interfaces. This chapter also describes briefly how to view and add instance variables that belong to an interface or interface component.

Advantages. Reusable interfaces and interface components have several advantages:

♦ Ensure consistency of the user interface across the application
♦ Capture common features of an interface in one area of code
♦ Cause the Designer to create code that you can reuse
Using a template to create an interface

The Designer allows you to use existing interfaces as a starting place for creating new interfaces. Think of existing interfaces as templates for new interfaces.

The Designer provides two types of templates, which are described in the following table:

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changeable</td>
<td>This kind of template has items and characteristics that you can change in the new interface. It provides a convenience to GUI programmers when they develop interfaces that share common features and behaviors.</td>
</tr>
<tr>
<td>Fixed</td>
<td>This kind of template has items and characteristics that you cannot change in the new interface. It enforces a windowing style across an application or a set of applications.</td>
</tr>
</tbody>
</table>

The following table describes the difference between the two types of templates:

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changeable</td>
<td>Copied to the new interface.</td>
<td>No change in the new interface.</td>
</tr>
<tr>
<td>Fixed</td>
<td>Inherited by the new interface.</td>
<td>Changes are reflected in the new interface.</td>
</tr>
</tbody>
</table>
**Changeable template:** For example, imagine that as you design the user interface for an application, you realize that many of the application’s forms contain a menu bar with File, Edit, and Help entries and OK and Cancel buttons. Instead of recreating each of these features for each form, create a changeable template with these features.

For each interface you create, select the changeable template from the Select Controller Type dialog box. Add new form items and remove any that are not relevant to the interface. Resize and customize the form.

**Fixed template:** As another example, imagine that you are designing a basic user interface that is going to be used in applications throughout your company. Perhaps you want to ensure that all main forms must at least have:

- OK and Cancel buttons
- The company name in italics in the form’s upper-right corner

You can create a template based on FixedItemsController. Instruct GUI programmers to use your template whenever they create new interfaces.
**Select Controller Type dialog box.** When you first open a new interface, the Select Controller Type dialog box appears:

The **Name** entry field enables you to define the name of the form (the form title is also set to the value that you define here). The **Create a new interface object like** list box lists the available interface templates. The **Description** box on the bottom contains a description of the selected template.

The following table describes the operations you can perform in the Select Controller Type dialog box:

<table>
<thead>
<tr>
<th>When you</th>
<th>The system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click a plus sign (+)</td>
<td>Displays the next level of items below the item next to the plus sign.</td>
</tr>
<tr>
<td>Click a minus sign (-)</td>
<td>Collapses the expanded list below the list item next to the minus sign.</td>
</tr>
<tr>
<td>Double-click a list item</td>
<td>Opens the Designer and displays the template you selected.</td>
</tr>
</tbody>
</table>
Making an interface into a template

Before you can create a template, you must create the interface.

Creating the interface

To create the interface:

1. Select File > New > Interface from the Desktop menu.
2. Select a template as described in the following table from the Select Controller Type dialog box:

<table>
<thead>
<tr>
<th>To create</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>A changeable template</td>
<td>Any controller except FixedItemsController</td>
</tr>
<tr>
<td>A fixed template</td>
<td>FixedItemsController</td>
</tr>
</tbody>
</table>

3. Click OK to open the Designer.
4. Add items to the interface and change characteristics as needed.

When you are done creating the interface, you can make it into a template.
Creating the template

To create a template from an interface:

1. Select **Controller > Change Options** from the Designer menu.

   The Controller Options Dialog box displays, as shown here:

   ![Controller Options Dialog](image)

   2. Enter a new name in the **Controller Name** entry field.

   3. Check the **Subclassable** check box.

   The **Explanation** entry field is enabled.

   4. Type a description of the controller in the **Explanation** entry field.

   5. Click **OK** to save your work or click **Cancel** to exit without saving.

   6. Select **File > Save as** to save the interface.

   7. Select **File > Exit** to exit the Designer.

When checked, the **Start cursors automatically** check box executes database statements automatically when the form containing the statement is opened. Uncheck the check box if you want the statement to execute when there is user action (such as a Button click).
Using a template

Once you have created the interface, you can use it as a template for another interface.

To use a template to create a new interface:

1. Click **New Interface** on the Desktop toolbar.
2. From the Select Controller Type dialog box, select the template on which you want to base your interface. Use the template that you created in the previous example. Note that you need to load the template file into the ObjectStudio environment.
3. Click **OK** to open the Designer.

   The template you selected displays in the Designer.
4. If you are creating an interface based on a fixed template, add items and characteristics. Remember that you cannot select or edit items that are part of the fixed template.
5. If you are creating an interface based on a changeable template, add or change items and characteristics.
6. Save the interface.

When you create a new interface based on the Controller, the system creates a subclass of the Controller class and an instance of the new subclass. Similarly, when you create an interface based on a template, the system creates a subclass of the template’s class and creates an instance of the new subclass.
Creating an interface component

The Designer allows you to create parts of interfaces, called interface components, which you can reuse. Once you define an interface component, you can have it display in the New Item dialog box. Then, to add the component to an interface, you treat it like any other form item.

For example, there might be several places in your application where you ask the user to select a font. You can create a font selector interface component that contains all form items necessary for choosing a font. Wherever the user needs to select a font in the application, place the font selector on an interface.

Creating a font selector

This section shows you how to build a font selector interface component. Later sections of this chapter build on this example.

The instructions for this example assume that you are familiar with the basics of creating form items. For more information about creating form items, see “3. Creating form items” on page 120.

The font selector interface component that you are going to build contains a Drop-down List Box where you can select a font. It also contains an Entry Field that displays an example of the font you select.
Getting started
To start creating the interface component:

1. On the Desktop, select the File > New > Component menu option.
   
   The Select Component Type dialog box appears:
   
   ![Select Component Type dialog box]
   
   2. Enter FontSelector into the Name field.
   
   3. Select ObjectStudio.InterfaceComponent from the Create new interface object like list box.
   
   4. Click OK.
   
   The Designer appears:
   
   ![Designer window]
   
   On the Designer menu bar, there is a Component menu instead of the Controller menu, because you are creating a component.
Creating form items
To create the form items:

1. In the Designer, select the FormItem > New Item menu option.
   The New Item dialog box appears.
2. Using the New Item dialog box, add a Drop-down List Box form item to the form.
3. Using the New Item dialog box, add an Entry Field form item to the form.
4. Resize and position the form items, as shown here:
Creating methods
To create two methods:
1. In the Designer, select the Tools > Methods menu option. The Method Editor appears.
2. Create the finishInit method:
   A. On the Method Editor, click the New button.
   B. In the Source edit area, type:
      ```smalltalk
      finishInit
      "Loads names of available fonts into the Drop-down List Box"
      | fonts |
      fonts := (OutputDevice getNamed: #Screen) availableFontFaces.
      dropdownListBox1 setInitialLabelsTo: fonts; setInitialValueTo: fonts.
      super finishInit.
      ```
   C. Click the Save button to save the method.
   D. Because the interface component inherits the finishInit method from its superclass, the system displays a message box asking if you want to subclass the method. Click OK.
3. Create the fontSelected method:
   A. On the Method Editor, click the New button.
   B. In the Source edit area, type:
      ```smalltalk
      fontSelected
      "Displays an example of a new font"
      self setValueTo:
      (Array with:
      (Array with: #FontFace with: dropdownListBox1 getValue)).
      entryField1 put:
      ('Example' asTextDefaultOptions: self value).
      ```
   C. Click the Save button to save the method.
4. Click the Close button to close the Method Editor.
Associating the method with the form items

The finishInit method executes when its controller initializes. Whenever a user selects a new font from the Drop-down List Box, the fontSelected method executes. The following procedure shows how to use the Event Editor to associate the fontSelected method with the Drop-down List Box. For more information about the initialization process, see “Initializing an interface” on page 306. For more information about the Event Editor, see “General events” on page 317.

To associate a method with the Drop-down List Box:

1. In the Designer, select Component > Events menu option.
   The Event Editor appears.

2. In the Event Editor, select dropdownListBox1 from the Sender objects list box.

3. Select changed: from the Event names list box.

4. Select FontSelector from the Specify event receiver list box.

5. Select fontSelected from the Messages list box.
6. Click the **Add** button.

The system adds information to the **Hooked receiver objects** list box and the left-hand **Messages** list box. The Event Editor appears as shown below:

7. Click **OK** to close the Event Editor.
Making the component reusable

You can add the interface component to the New Item dialog box or make it a reusable interface component in the Select Controller Type dialog box.

To make the component reusable:

1. In the Designer, select the **Component > Change Options** menu option.

The Interface Component Options dialog box appears:
2. To make the interface component display on the New Item dialog box, make these changes to the fields in the Item Palette topic box:
   A. In the Icon file entry field, change the name of the icon file if you want the icon that displays in the New Item dialog box to be different from the default.
   B. Enter a description for the interface component into the Description field. In this example, the Description field already contains the text “FontSelector.”
   C. Enter the name of the interface component into the Name field. In this example, the Name field already contains the text “FontSelector.”
   D. Select the Visible In Item Palette check box.
3. Optional: To make the interface component display in the Select Component Type dialog box so that you can create subclasses of it, make these changes in the Subclasses topic box:
   A. Select the Subclassable check box.
      The system enables the Explanation entry field.
   B. Enter a description of the interface component into the Explanation entry field.
4. Optional: To change the events that the interface component generates, make these changes in the Supported Events topic box:
   A. Delete a method by selecting it and clicking the Delete button.
   B. Add a method by entering its name into the entry field at the bottom of the form and clicking the Add button.
5. Click OK to save your changes or click Cancel to exit without saving.
Saving and testing your work

To save and test the interface:

1. In the Designer, select the **File > Save as** menu option.
   - The Save As dialog box appears.

2. Use the Save As dialog box to select a folder, to name the file fontsel.cls, and to save the file.

3. In the Designer, select the **File > Test interface** menu option.

4. Select a font from the **Font Selector** drop-down list box (for example, choose Arial).
   - The interface appears:

5. In the Designer, select the **File > Edit interface** menu option to continue using the Designer.

You have finished creating the font selector interface component. The next section shows you how to add the component to an interface.

For more information about associating methods with form items, see “Assigning methods to form items” on page 150. For more information about action events and other types of events, see “Working with events” on page 310.
Adding an interface component to an interface

This section describes how to add an interface component to an interface.

**Overview of procedure:** Once you create an interface component and add it to the Item Palette as you did in the previous example, an icon for it displays on the New Item dialog box. By default, the icon is shaped like a puzzle piece.

To use an interface component in an interface follow these stages:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create the new interface and display the New Item dialog box.</td>
</tr>
<tr>
<td>2</td>
<td>Follow the procedure for creating a form item to create an instance of the interface component you defined.</td>
</tr>
<tr>
<td>3</td>
<td>Add other form items to the interface.</td>
</tr>
<tr>
<td>4</td>
<td>Position, resize, and customize form items or the interface component.</td>
</tr>
</tbody>
</table>

**What happens:** When you include an interface component in an interface, the system treats the interface component like one unit. For example, you can move the entire interface component, but you cannot move one form item in the component independently of the other form items.

**Smalltalk details:** When you create an interface component, the system creates a subclass of the InterfaceComponent class. Each time you use the interface component, the system creates an instance of the new subclass. This approach enforces absolute consistency wherever you use the interface component.

**When you change an interface component:** Imagine that you create an interface component, include it in a few interfaces, and then change the interface component. To update the interfaces that use the component, reload the class file associated with each interface. When you reload the class file, the system updates the interfaces to reflect the changes you made to the interface component.
Using the font selector in an interface

This section shows you how to include the font selector in an interface. When you are finished with this section, the interface will contain the font selector and a List Box that lists employee names.

Including the font selector interface component

To include the font selector interface component in an interface:

1. To create a new interface:
   A. On the Desktop, select the File > New > Interface menu option.
      The Select Controller Type dialog box appears.
   B. In the Select Controller Type dialog box, enter Employees into the Name field.
   C. Select ObjectStudio.Controller from the Create new interface object like list box.
   D. Click OK.

2. In the Designer, select the FormItem > New Item menu option.
   The New Item dialog box displays.
3. On the New Item dialog box, double-click the FontSelector icon.

Recall that the items in the dialog box are alphabetized. If you select an item, its name displays below the **Name** entry field.

The system creates a new font selector and adds it to the Employees form. To move the font selector, select the form item and then move it as you would move any normal form item.

4. Using the New Item dialog box, add a **List Box** form item to the Employees form.

5. Link the **List Box** form item to the Employee table:
   A. Double-click the **List Box**.
      
      The Form List Box Options dialog box appears.
   B. Click the **Link** button.
      
      The Link Hookups dialog box appears.
   C. In the first three List Boxes, from left to right, select **Table**, **Employee**, and **Name**.
   D. Click **OK** to return to the Form List Box Options dialog box.
   E. Click **OK** to return to the Designer.
Saving and testing the font selector

To save and test the font selector:

1. In the Designer, select the **File > Save as** menu option.
   The Save As dialog box appears.

2. Use the Save As dialog box to select a folder, to name the file fontinit.cls, and to save the file.

3. In the Designer, select the **File > Test interface** menu option.
   The interface appears:

![Image of Employees interface]

4. In the Designer, select the **File > Edit interface** menu option to continue using the Designer.

You have finished creating the Employees interface. In “Completing the font selector” on page 320, you will use the **Event Editor** to change the font in the List Box whenever you select a new font.
Creating new interface form items

This section shows you how to create a version of a Number Field that displays:

♦ The dollar sign ($) in front of any data
♦ Positive numbers in blue
♦ Negative numbers in red

Some of the steps assume that you know how to use the Text Editor or the Class Browser to create new classes and methods. If you need help with either of these tools, refer to the *ObjectStudio User’s Guide*, P40-3807.
Creating the form item

To create the form item, use the Class Browser or Text Editor to:

1. Create a new class using the following information:

   Class Name: FormMyNumber
   Superclass: FormNumber
   Instance variables: none
   Class variable: defaultWndProc
   Pool dictionaries: none

   Note that you must define the defaultWndProc class variable for every form item that you create. The variable refers to the default window procedure that the Designer uses. If you do not define this variable, an error occurs.

2. Save the class to a file (for example, mynumfi.cls).

3. Create the following class method, which provides a default maximum length, in characters, for the field. (The maximum number you can use is 99.)

   ```
   def defaultMaximumLength
   ^ 20.
   ```

4. Create the following class method, which provides a name for the item in the New Item dialog box:

   ```
   def description
   ^ 'My Number Field'.
   ```

5. Create the following class method, which specifies the file name for the icon to use for the item in the Designer’s New Item dialog box. Ensure that the pathname is complete enough for your system.

   ```
   def iconFileName
   ^ 'demo\tutorial\dollar.ico'.
   ```

6. Define the following instance method, which specifies the controller item class that corresponds to this form item:

   ```
   def controllerItemClass
   ^ MyNumberCtrl.
   ```
Creating the controller item

To create the corresponding controller item, use the Class Browser or Text Editor to:

1. Create a new class using the following information:
   - Class name: MyNumberCtrl
   - Superclass: NumberCtrl
   - Instance variables: none
   - Class variables: none
   - Pool dictionaries: none

   You do not need to define any class methods for this controller item.

2. Save the class to a file (for example, mynumci.cls).

3. Create the following instance method, which is called when the item receives a new value and needs to be redisplayed. The method converts the value in the Number Field into a NumericText item and formats it.

```plaintext
hitKeyKeyName: key
| returnVal |
returnVal := super hitKeyKeyName: key.
val := self getValue.
(val isKindOf: NumericText) ifFalse: [
   (val isKindOf: Number) ifFalse: [
      decimalChar == nil ifTrue: [
         decimalChar := (0 asTextOptions:
            format)
      decimalChar.
   ].
   val := val asString.
   decimalChar ~~ $. ifTrue: [
      val := val copySubstituting: ').' for: decimalChar.
   ].
   val := val asNumber.
   val == nil ifTrue: [
      val := self defaultInitialValue.
   ].
   format := Array with: {#Currency #USA}.
   val >= 0 ifTrue: [
      format add: #Blue.
   ] ifFalse: [
      format add: #Red.
   ].
   val := val asTextOptions: format.
   self setText.
]^ returnVal.
```
Using the new user interface item

To use the new user interface item:

1. Load the files mynumfi.cls and mynumci.cls into ObjectStudio, if they are not already loaded.

2. Open the Designer and select **FormItem > New Item** to display the New Item dialog box.
   
   Note that the My Number Field item displays in the dialog box. Its icon is a dollar sign.

3. Add the new item to a form.

4. Save and test the interface.

   A dollar sign always precedes the data and the color of the data changes depending on whether the value is positive or negative.
Working with instance variables

As you work with the Designer, the system often creates instance variables so that you can refer to controller items in code. In some cases, instance variables are not created for you, or you may forget to create an instance variable for a form item.

The Designer provides the Instance Variables dialog box that allows you to view, add, or delete the instance variables that you have defined for an interface or an interface component. To display the dialog box, select Tools > Variables from the Designer menu. The Instance Variables dialog box displays, as shown here:
Adding an instance variable

To add an instance variable:

1. Click Add on the Instance Variables dialog box.

   The String Dialog box displays, as shown here:

   ![String Dialog](image)

   2. Type the name of the instance variable in the entry field. Remember that by Smalltalk convention, instance variables begin with a lowercase letter.

   3. Click OK.

   The system returns focus to the Instance Variables dialog box and adds the new variable to the Variables list box.

Deleting an instance variable

To delete an instance variable:

1. Select the variable.

2. Click Delete.

   The instance variable is removed from the Variables list box.

Exiting the Instance Variables dialog box

To exit the dialog box, click Close.
Multiple document interface

Multiple Document Interface (MDI) is a user interface style that allows an interface to support multiple documents within an MDI main form. An MDI form can contain a menu bar, toolbars, status lines, and subforms. The subforms, which are clipped to the parent, can contain form items. Microsoft Word is an example of an MDI application.

Creating an MDI application

To create an MDI application:

1. Click New Interface on the Desktop toolbar or select File > New > Interface to create a new interface and to open the Designer.

2. Select ObjectStudio.MDIWindowsController from the Select Controller Type dialog box.

The Designer displays the main MDI form.

By default, the MDI main form has a menu bar that includes a File, Window, and Help menu. You can add toolbars, status lines, menu options, and subforms to the main MDI form. You can add any form item or menus to subforms.

Note that a subform is in class MDIWindowsChildForm and not in class SubForm.
Working with MDI menus

You can add menu options to the default menu on an MDI parent form, and you can create a menu bar for each of its subforms. When a subform has focus, its menu replaces the parent's menu.

To create a menu for an MDI form or one of its subforms:

1. Select Form > Form name from the Designer menu, where Form name is the form for which you want to create a menu.

2. Select Tools > Menus from the Designer menu.

The Menu Setup dialog box displays, as shown here:

Use the Menu Setup dialog box to add menu items to the existing pull-down menus or create new pull-down or pop-up menus. For more information about the Menu editor, see “Menu editor” on page 75.
Merging menus

You can merge the menu of a child form with the menu of the parent form (rather than having the menu of the child form replace the menu of the parent form). To merge menus, subclass the openMenusForForm: init: method in class Controller; for example:

``` Smalltalk
openMenusForForm: aForm init: fInitMenu
    aForm == (formDict at: #MDIChileOne) ifTrue: [aForm mergeMenu1: 
        (aForm parent menuBar) menu2: (aForm menuBar)
        at: 2 remove: {#ParentMenuToBeRemoved ..}. 
        ^ self.
    ].
    "must continue this message for its super if not merging menu"
    super openMenusFormForm: form init: fInitMenu.
```

For an example of how to merge menus, see the file tmdimenu.cls in the Sample folder.
Adding a Window menu to an MDI form

A standard native Window menu displays on an MDI form when it is created with the Designer.

To add a Window menu to an MDI form programmatically, use the following code (code in italics is code that you supply):

```lisp
ctrlItem := controller add: #anyName
class: FormSubMenu
position: nil
options: nil
form: form
text: '&Window (optional text)'.

ctrlItem setInitialLabelsTo:
(FormWindowsMenu mfcWindowsMenu).

ctrlItem setInitialValueTo: #(ValueArray).
```

Note that the length of the ValueArray must equal the length of the LabelArray.

Customizing the Window menu

You can add and delete labels from the Window menu. However, to maintain the default behavior of the menu, do not change the labels of the standard menu.

For example, the original array of labels is:

```text
{"&Cascade' 'Tile &Horizontal' 'Tile &Vertical
 'Arrange &Icons'}
```

You can remove the last two items in the array and insert others to form the following valid array of labels:

```text
{"&Cascade' 'My Label 1' 'Tile &Horizontal'
 'My Label 2'}
```

The following is not a valid label array because Window menu labels were changed:

```text
{"&Cascade' 'Tile &Hor' 'Tile &Ver' '&Icons'}
```
Calling Window menu methods

When you select items from a predefined Window menu, the system directly calls the Windows Microsoft Foundation Class (MFC) Application Programming Interface (API). ObjectStudio also provides methods in class MDIForm that do the same functions as the MFC API functions. These methods have names such as cascadeChildren and arrangeIcon. Call these methods to give access to Window menu functions from a part of the user interface other than a predefined Window menu.

Adding a Window menu to other forms

If you add a Window menu to forms other than MDI forms, ObjectStudio provides a template Window menu that you can modify any way you want. When the user selects an item from a non-MDI Window menu, ObjectStudio does not call native MFC API code.
5. Creating a workplace application

What is a workplace application?

Workplace applications offer the user an intuitive, easy-to-learn work environment in which icons represent real-world objects. The user manipulates these objects with the mouse.

How ObjectStudio helps you. To assist you in creating workplace applications, ObjectStudio provides:

♦ Predefined workplace objects
♦ Method source code that defines the behavior of objects
♦ The underlying object-oriented foundation that allows you to design and build a workplace application

Online demo. ObjectStudio provides an example of a workplace application. You can experiment with using it. You can also examine its code to learn about aspects of workplace applications that are beyond the scope of this section.

To load the workplace application into the ObjectStudio environment:

1. Select Load application from the Desktop pop-up menu.
2. Select Workplace Demo from the Applications dialog box.
3. Click Load to load the application.
4. Click Close to close the Applications dialog box.
   The icons for the Workplace Demo display on the work area of the Desktop.
5. Double-click the Personnel Workplace icon and experiment with the applications.

A list of class files that implement the workplace demo is in the file wpdemo.txt, which is located in the ObjectStudio installation folder.
Using a workplace application

The following table describes the mouse and keyboard actions that the user can take once you define a workplace application:

<table>
<thead>
<tr>
<th>To</th>
<th>Take this action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select an object</td>
<td>Click the object once with mouse button 1. You also can use the keyboard arrow keys to move the selection to the next icon.</td>
</tr>
<tr>
<td>Select several objects</td>
<td>Select one object. Press the CTRL key while you select other objects. You also can press and hold mouse button 1. As you move the mouse, the system draws a rectangle. When you release the mouse, all objects enclosed by the rectangle are selected.</td>
</tr>
<tr>
<td>Open an object</td>
<td>Double-click the object with mouse button 1. You also can select the object and press ENTER.</td>
</tr>
<tr>
<td>Move an object (Drag)</td>
<td>Press and hold mouse button 2 while you move the mouse. When the object is where you want it, release the mouse button.</td>
</tr>
<tr>
<td>Copy an object</td>
<td>Press and hold the CTRL key while you drag the object.</td>
</tr>
</tbody>
</table>
Workplace form items

The Designer provides three types of form items with predefined workplace application behavior. The following table describes these items:

<table>
<thead>
<tr>
<th>Form item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workplace List Handler</td>
<td>Container that you can use to store and organize other objects. By default, it displays its contents in icon format. It can also display as text or as an icon with text. The user can operate on individual objects or on the entire container.</td>
</tr>
<tr>
<td>Workplace Object Handler</td>
<td>Repository for information. It combines data and code that manages the data.</td>
</tr>
<tr>
<td>Workplace Object Template</td>
<td>Allows you to create templates for new Workplace Object Handlers.</td>
</tr>
</tbody>
</table>

The Designer includes the three workplace objects on the New Item dialog box. Use the Workplace List Handlers and Workplace Object Handlers the same way you use any form item. When you define a new Workplace Object Template, you must declare characteristics specific to that template.

**Dynamic option:** The Dynamic check box on the Workplace Object Handler and Workplace Template Definition dialog boxes allows you to set up the object so that the text of its icon can change. For example, if you are working with a form item that displays information about an employee, you can set the object so that its icon displays the name of the employee associated with the Workplace Object Handler.

When you check the Dynamic check box, the system sets the dynamicName instance variable to true. This value of dynamicName in turn notifies the system to send a getValue message to get the name of the workplace object.

The getValue method works with the name method, which is defined in the workplace object class. The name method returns the value that the system will use for the object name. To change the name, send the setDynamicNameTo: message to the workplace object.

The system sets the dynamic name during initialization and never recalls the method. To change the dynamic name, your code must send a getValue message to the object.
Creating a workplace application

This section shows you how to create a workplace window that contains a folder and an editor. You will create simple objects that you can collect in the folder and edit with the editor. Use the following figure as a guide as you complete the instructions:

The directions for this example assume that you understand the basics of using the Designer. If you need more detailed explanations, see “2. Creating forms” on page 45 and “3. Creating form items” on page 120.

Sample code. The code for the finished application is in the file mywp.cls, in the ObjectStudio tutorial folder. You can load the file from the tutorial folder and examine it.
Creating the main form

To create the main form of the workplace application:

1. On the Desktop, select the **File > New > Interface** menu option.
   The Select Controller Type dialog box appears.

2. In the Select Controller Type dialog box, enter the following into the **Name** entry field:
   `MyWorkplace`

3. Select `ObjectStudio.Controller` from the **Create new interface object like** list box.

4. Click **OK**.

Adding form items

To add form items to the form `MyWorkplace`:

1. In the Designer, select the **FormItem > New Item** menu option.
   The New Item dialog box appears.

2. Use the New Item dialog box to create a **Workplace List Handler** form item named `Folder`.

3. Use the New Item dialog box to create a **Workplace Object Handler** form item named `myObject`.

4. Use the New Item dialog box to create a **Workplace List Handler** form item named `myEditor`.
Changing form items

To change the icon for myObject and provide a default value:

1. Double-click the myObject icon.

The WP Object Handler Definition dialog box appears:

2. Select Info from the Icon Names list box to change the icon for the object.
3. Select Special from the Value radio buttons, if it is not already selected.
4. Enter 5 into the Value Specification entry field, in order to specify a default value for the object.
5. Click OK to save your changes and close the dialog box.

To change the icon for the myEditor object:

1. Double-click the myEditor icon.

The WP List Handler Definition dialog box appears.

2. Enter Program into the Icon Specification entry field to change the icon for the object.
3. Click OK to save your changes and close the dialog box.
Creating the editor form

Create an editor form to assign to the Workplace Object Handler. On this form, create an Entry Field that displays the value of the workplace object and allows the user to change the value. Later, you will implement methods that activate the window when a workplace object with a number value is dropped on the myEditor icon. The finished Object Editor is shown here:

To create the Object Editor form:

1. In the Designer, select the Form > New form menu option.
   The String Dialog dialog box appears.
2. In the String Dialog dialog box, enter the following into the Enter new form name entry field:
   Object Editor
3. Click OK.
Adding form items

Add the following form items to the Object Editor form. Use the form shown in “Creating the editor form” on page 283 to help you position the form items.

1. Create a **Static Text** form item. Set its **Name** to valueText and its **Text** to Value:.
2. Create a **Number Field** form item. Set its **Name** to value.
3. Create a **Button** form item. Set its **Name** to ok and its **Text** to OK.
4. Create a **Button** form item. Set its **Name** to cancel and its **Text** to Cancel. Edit the **Cancel** button:
   A. Double-click the **Cancel** button to open the Form Button options dialog box.
   B. Select **Close Form** from the drop-down list box that has an initial value of **Custom**.
   C. Click OK.

Creating an instance variable

By default, the Designer does not create an instance variable for Workplace Object Handlers. However, in this example you need to create one additional instance variable to hold the value of the workplace object.

To create an instance variable:

1. Select **Tools > Variables** from the Designer menu to display the Instance Variables dialog box.
2. Click **Add**.
3. Type the following variable name in the **Enter instance variable** entry field:
   
   wpObj

4. Click **OK**.
5. Click **Close** to save your changes and close the Instance Variables dialog box.
Subclassing the accepts:in: method

In this section, you create a method to specify that the Object Editor accepts workplace objects with number values. In other words, the method allows the user to drag an object with a number value to the Object Editor.

The accepts:in: method is defined in class ObjectStudio.Controller. Its default behavior for this class is to accept nothing. Subclass this new method to specify that the myEditor object will accept objects with number values.

To subclass the method:

1. Select Tools > Methods from the Designer menu to open the Method Editor.
2. Click New.
3. Type the following method in the Source edit area:
   ```plaintext
   accepts: anObj in: fm
   fm == (formDict at: #'Object Editor')
   ifTrue:
     ^ (anObj getValue) isKindOf: Number.
   ^ false.
   ```
4. Click Save to save the method.
5. Click OK to confirm that you want to subclass the method.

Do not exit the Method Editor.

You do not need to assign the accepts:in: method to an object. The system calls it automatically when an icon is dragged over the myEditor icon.

When the user drags an object onto an icon, an event occurs that triggers the accepts:in: message and sends it to the object represented by the icon.

The code that you just wrote states that if the object you are dragging has a number value, then the myEditor object accepts it. Otherwise, the method returns false and the system displays an Illegal icon (a circle with a bar) and does not allow the operation.
Creating the in:drop:at: method

Define a method to display the value of the dropped object in the Object Editor’s Value entry field. Again, this method does not need to be assigned to any object. The system automatically calls it when the user drops an object on the Object Editor.

To create the in:drop:at: method:
1. Click New on the Method Editor.
2. Type the following method in the Source edit area:

   in: fm drop: anObj at: pos
   fm == (formDict at: #'Object Editor')
   ifTrue:
       [ value put: (anObj getValue).
         wpObj := anObj.
         fm open. ].

3. Click Save to save the method.
   Do not exit the Method Editor.

Creating the doOk method

The doOk method defines the behavior of the OK button. When the user clicks OK, this method stores the current value of the object and closes the form.

To create the doOk method:
1. Click New on the Method Editor.
2. Type the following method in the Source edit area:

   doOk
   wpObj setValueTo: (value getValue).
   (formDict at: #'Object Editor') close.

3. Click Save to save the method.
4. Click Close to exit the Method Editor.

Assigning the doOk method

To assign the doOk method to the OK button:
1. Double-click the OK button to display the Form Button Options dialog box.
2. Click Methods to display the Method Assignment dialog box.
3. Select doOk from the Available Methods list box.
4. Click Copy to add the method to the Assigned Methods list box.
5. Click OK to save your changes and close the Method Assignment dialog box.
6. Click OK to close the Form Button Options dialog box.
Assigning a return key method

In this section, you assign a return key method. When the user presses ENTER while focus is on the value field or on the OK button, the system calls the doOk method.

To assign the return key method:
1. Set focus to the Object Editor form.
2. Select Form > Return key methods from the Designer menu.

The Return Key Method Assignment dialog box displays, as shown here:

3. Select OK and value from the Items list box.

   **Recall that to select more than one item from a List Box, press the CTRL key while you select subsequent items.**

4. Select doOk from the Available Methods list box.
5. Click Copy to add the method to the Assigned Methods list box.
6. Click OK to save your changes and close the Return Key Method Assignment dialog box.
Assigning a form to an object handler

To assign the Object Editor form to the myEditor object handler:

1. Set focus to the MyWorkplace form.
2. Double-click the myEditor icon.
   
   The WP Object Handler Definition dialog box displays.
3. Check the Form radio button in the Value group at the bottom of the window.
4. Type the form name Object Editor in the Value Specification entry field.
5. Click OK to save your changes and exit the dialog box.
6. Select Controller > Change Options to name the workplace application.

   Type a new name in the Controller Name entry field.
7. Select File > Save as to save the interface.
8. When prompted to save the interface, type the file name mywp.cls.
Testing the interface

To test the interface:

1. Select File > Test interface from the Designer menu.
2. Double-click the Folder icon.
   A list handler window opens, displaying the myObject icon.
3. Move myObject out of the folder, back to the workplace window, and close the list handler window.
4. Drag the myObject icon onto the myEditor icon.
5. Double-click the myEditor icon to open the Object Editor object handler window. It displays the current value of the dropped object.
6. Type a new value for the object; then click OK.
7. When you drop the object onto the editor again, the window displays the new value.
8. Create a copy of the myObject icon:
   A. Select the myObject icon.
   B. Press and hold the CTRL key and press mouse button 1 to drag the myObject icon.
   C. Edit the copy to change the value.
   D. Edit the original object.
9. The original object maintains its value; the value of the copy is changed.
10. Select File > Edit interface to return to Edit Interface mode.
**Adding a pop-up menu**

This section shows you how to use pop-up menus.

**Creating the pop-up menu**

To edit the interface and create a pop-up menu:

1. Select **Tools > Menus** from the Designer menu.
2. In the **Caption** entry field, type testMenu.
3. Check the **Not in Menu Bar** check box.
4. Click **Add** or **Insert**.
5. Create pop-up menu items:
   A. Type one in the **Caption** entry field and click **Add**.
   B. Type two in the **Caption** entry field and click **Add**.
   C. Type three in the **Caption** entry field and click **Add**.
6. Select one in the **Menu Definition** list box and press the right arrow to demote the item to the pop-up menu. Repeat this step for the two item and the three item.
7. Click **OK** to save your changes and exit the Menu Editor.

**Assigning the pop-up menu with the Designer**

You can use the Designer to attach pop-up menus to Workplace Object Handler items.

To attach the testMenu pop-up menu to the myObject object handler:

1. Double-click the myObject item to open the WP Object Handler Definition dialog box.
2. Select testMenu from the **Pop-up Menu** list box.
3. Click **OK** to close the dialog box.
Assigning the pop-up menu programmatically

The Designer does not support assigning pop-up menus to Workplace List Handler objects or other form items.

To assign a pop-up menu to a Workplace List Handler:

1. Select **Tools > Method** from the Designer menu to display the **Method Editor**.
2. Click **New**.
3. Type the following method in the **Source** edit area:
   ```small
   finishInit
   (cItemDict at: #Folder)
   setPopupMenuTo: (cItemDict at: #testMenu).
   ```
4. Click **Save** to save the method.
   You are asked if you want to subclass the method. Click **OK** to subclass the method.
5. Click **Close** to close the Method Editor.

**Saving and testing the pop-up menu**

To save and test the interface:

1. Select **File > Save** from the Designer menu.
2. Select **File > Test interface** from the Designer menu.
3. Select the Folder object and press mouse button 2 to display the pop-up menu.
4. Select **File > Edit interface** to return to edit mode.
5. Select **File > Exit** to close the Designer.

**When pop-up menus are not available:** If you assign a pop-up menu to a workplace object that is in a Workplace List Handler, the user has access to the object’s pop-up menu only when the window is open in icon view. When the window is open in text or table view, the pop-up menu is not available.
Creating a Workplace Template Object

When you create a workplace template, you can define common characteristics of an object as a starting place for new objects. Once you define a template, you can create a new object by dragging the template.

Creating a new class to define the template

Before you use the Designer to create the Workplace Template Object, you need to create a new class that defines the object.

To create the new class:
1. Select Tools > Class Browser from the Desktop menu to open the Class Browser.
2. Select Object from the list of classes.
3. Select Class > New from the Class Browser menu to display the Class Definition dialog box.
4. Type Data as the class name and confirm that Object is the superclass.
5. Click OK to save your changes and exit the dialog box.
   The Save As dialog box displays.
6. Type a file name in the File name entry field (for example, datanum.cls).
7. Click Save to save the file and close the Save As dialog box.
8. Select the Class radio button from the Class Browser.
9. Select Method > New from the Class Browser menu.
10. Type the following method text in the Source edit area:

    templateValue
        | wpObj |
        wpObj := WorkplaceObject new.
        wpObj setNameTo: #'Number Object'.
        wpObj setIconSourceTo: #SystemIcon.
        wpObj setIconSpecTo: #Info.
        wpObj initialize.
        wpObj setValueTo: 0.
        ^ wpObj.

11. Select Method > Save from the Class Browser menu.
12. Select Class > Exit to exit the Class Browser.
Creating a workplace template

To substitute a Workplace Template Object for the Workplace Object Handler named myObject:

1. Select File > Load file to open the MyWorkplace file mywp.cls.
2. Select the MyWorkplace icon and press mouse button 2 to display the pop-up menu.
3. Select Edit from the pop-up menu to open the Designer.
4. Select the myObject Workplace Object Handler.
5. Select FormItem > Delete Item(s) from the Designer menu.
6. Create a WpObj template:
   A. Select FormItem > New Item to display the New Item dialog box.
   B. Type Number Template in the Name entry field.
   C. Double-click the icon for a Workplace Template Object to create it.

7. Edit the Number template:
   A. Double-click the new form item to open the WP Template Definition dialog box.
   B. Select the Global radio button.
   C. Type Data (the name of the class you created) in the Value Specification entry field.
   D. Click OK to save your changes and exit the dialog box.
8. Select File > Save to save the file.
Testing the workplace template

To test the workplace template:

1. Select File > Test interface.

2. Select the Number Template object, press mouse button 1, and drag the mouse to an empty area of the workplace form to create a new Number Object when you release mouse button 1.

   The icon changes from the QuestionMark to the Info icon assigned to wpObj in the class method you created.

3. Then:
   
   A. Drop the Number Object onto the myEditor icon.
   
   B. Verify that the default value is 0.
   
   C. Type a new value.
   
   D. Click OK.

4. Drag the Number Object to myEditor again to verify that the value was saved.

5. Create another Number Object and edit it. Verify that the two objects maintain their separate values.

6. Close the interface when you are done testing.
6. Using the Model Editor

Introduction

This chapter describes the Model Editor, with which you can create variables, then define and test relationships between them using mathematical formulas.

Description of the Model Editor

The Model Editor allows you to:

♦ Define variables and set their start, stop, and step values
♦ Create mathematical formulas based on variables
♦ View variable names in a convenient format so you need not recall, for example, whether profit equals price * units or sales * units
♦ Define and test relationships between variables and formulas before using them in an interface
♦ View the variables as graphs
♦ Link variables to interfaces in the ObjectStudio Designer
♦ Save models as class files and revise them as needed

The Model Editor displays information dynamically, as described in the following table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Names and values of constants.</td>
</tr>
<tr>
<td>Formula text</td>
<td>Constants or expressions for a variable selected from the Inputs, Intermediates, or Outputs boxes.</td>
</tr>
<tr>
<td>Intermediates</td>
<td>Variables that are the results of formulas and are used as components of other formulas.</td>
</tr>
<tr>
<td>Outputs</td>
<td>Variables resulting from formulas and not used elsewhere in the model.</td>
</tr>
<tr>
<td>Indexed Values</td>
<td>Vectors associated with the selected variable.</td>
</tr>
<tr>
<td>Undefined Vars</td>
<td>Variables that are not associated with either values or formulas.</td>
</tr>
<tr>
<td>Reference</td>
<td>Cross-references showing the selected variable at the right, related variables at the lower left, and related formulas at the lower right.</td>
</tr>
</tbody>
</table>
Opening the Model Editor

To open the Model Editor: on the Desktop, select the File > New > Model menu option.

The system opens the Model Editor, assigns a numeric identifier to the new model, and displays the Variable Definition dialog box, as shown here:

To use the Model Editor, first define the names of your variables; then assign values for them, if appropriate. Next, use the variables in formulas. You can have the system calculate results immediately or on demand.

After defining variables and formulas, you can design a graph. Variables and graphs can be used in the Designer. For more information about the Designer, see “1. Introducing the Designer” on page 11 and “2. Creating forms” on page 45.
Model Editor menu bar: The Model Editor menu bar contains options that allow you to manage the current file, the current model, its variables, and its graphs. The following table describes the options on the Model Editor menu bar:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Contains options to load and save files. Also contains an option that allows you to change the name of the model.</td>
</tr>
<tr>
<td>Variable</td>
<td>Allows you to define and manage variables.</td>
</tr>
<tr>
<td>Calculate</td>
<td>Provides an option for calculating formulas and models.</td>
</tr>
<tr>
<td>Graph</td>
<td>Contains options that let you set up and display a graph.</td>
</tr>
</tbody>
</table>

Defining variables

ObjectStudio allows you to define several variables at one time or define variables one at a time. After creating variables, you can define them as constants or as expressions.

Creating variables

When you open the Model Editor, the Variable Definition dialog box displays automatically.

To create variables:

1. Type a name in the Variable Definition dialog box (for example, type originalCost).
2. Click the Apply button or press ENTER. The variable displays in the Inputs list box.
3. Repeat steps 1 and 2 as needed to define all variables for your model (for example, type currentValue; then type profit).
4. Click Cancel to close the Variable Definition dialog box.

To create additional variables, either:

- Select Variable > New if you need to create additional variables.
- Select Variable > Quick New to create one new variable. After you type the name and click Apply or press ENTER, the dialog box closes automatically.
Assigning values to variables

Assign values to variables to establish starting points or to enter test data for your model. You can assign constants or expressions to any variable. To assign a value to a variable:

1. Select a variable in the Inputs list box. For example, select profit.
2. Type a constant or an expression in the Formula text field. For example, type:
   \[ \text{currentValue} - \text{originalCost} \]
3. Press ENTER.
4. Highlight another variable. For example, select originalCost.
5. Type a constant or expression in the Formula text field. For example, type 100000.
6. Press ENTER.
7. Select a calculation option as appropriate:
   A. Check the Automatic calculation check box to recalculate the model after modifying a variable or formula. (Use the Calculate menu if you disable automatic calculation.)
   B. Check the External calculation display check box to update the Model Editor (if it is open) when you modify a linked variable in an interface. If disabled, the system updates the Model Editor only when you make it the active form.
8. Optional: Set up or display a graph of your model, as described in the following section.
Setting up a graph

The graph feature in the Model Editor displays your variables and formulas in line graphs or in grouped or stacked bar graphs. You can modify the horizontal axis by changing the start, stop, and step values.

To define a graph:

1. Select **Graph > Setup**.
   
   The Set up Model1 Graph dialog box appears:

   2. Select an input and an output variable. (Output variables display on the Y-axis, and the input variable displays on the X-axis.)
      
      You can select more than one output variable, but the graph only makes sense if all selected output variables depend on the same input variable.

   3. Enter start, stop, and step values to establish the beginning, ending, and incremental values for the X-axis, respectively.

   4. Select a graph type radio button.

   5. Click **OK** to display the graph.

   6. Click any area outside the graph display to return to the Model Editor.

   7. *(Optional)* Select **Graph > Display** to redisplay the graph.
Linking models and graphs to interfaces

You can link a model to a Number Field or to a Chart in an interface. You can use the Update link option to link to input variables only.

For more information about linking and the Designer, refer to the *ObjectStudio OLE User's Guide*, P40-3805.
7. Programming graphical user interfaces

Visual interface objects

Standard windows are represented by instances of the Form class. A form in the context of ObjectStudio is just another word for window and is a much more general concept than a data-entry form.

More specialized windows are represented by subclasses of Form. Dialog boxes have a DialogBox class, and message boxes have a MessageBox class. Child windows that are created inside the main window are instances of SubForm and child windows that are created inside an MDI window are instances of MDIWindowsChildForm.

Controls are derived from a FormItem class. For example, there is a FormButton class, a FormList class, a FormString class, and so on, that are all subclasses of the FormItem class.

Each Form and FormItem has a name. That is, a symbol used to reference it when programming. For example, a form can be called #form1, and a Button can be called #okButton.
A Form instance maintains a list of the FormItems displayed on it. The Form instance owns the FormItems, as it is responsible for creating them when the form is opened, and for destroying them when the form is closed, as shown here:

The list of items for a form is kept in an array instance variable called items.
Form and form item geometry

ObjectStudio uses tenths of millimeters (0.1 mm) for all measurements related to screen-coordinate values. This scheme helps you maintain device independence as you develop an application.

For more information about accommodating differences in monitor resolution, refer to the explanation of the −x and −y command line parameters in the ObjectStudio User’s Guide, P40-3807.

Form coordinate system. A form’s origin (the (0,0) coordinate) is in the upper-left corner of the screen. The form’s border, title bar, and menu bar are not included in the form’s coordinate system. The following figure illustrates a form’s coordinate system:
**Form item coordinates:** A form item has an associated rectangle, an array of four elements, which describes the form item’s size and its location on the form. The rectangle contains four of the six values described in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Is the distance between</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>The form item’s bottom edge and top edge.</td>
</tr>
<tr>
<td>Width</td>
<td>The form item’s left edge and right edge.</td>
</tr>
<tr>
<td>x1</td>
<td>The left edge of the form and the left edge of the form item.</td>
</tr>
<tr>
<td>x2</td>
<td>The right edge of the form item and the right edge of the form.</td>
</tr>
<tr>
<td>y1</td>
<td>The top edge of the form and the top edge of the form item.</td>
</tr>
<tr>
<td>y2</td>
<td>The bottom edge of the form item and the bottom edge of the form.</td>
</tr>
</tbody>
</table>

The following figure illustrates each of these values:

By default, a form item’s rectangle is \{ x1 y1 width height \}.
**Effect of resizing a form:** Within the Designer, you can set options that determine whether a form item moves or is resized when its form is resized.

The following table describes the effect of setting each option (or combination of options) on a form item’s array of options and on its rectangle:

<table>
<thead>
<tr>
<th>Option</th>
<th>Adds to option array</th>
<th>Sets rectangle to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertically Moveable</td>
<td>#VMove</td>
<td>{ x1 y2 width height }</td>
</tr>
<tr>
<td>Horizontally Moveable</td>
<td>#HMove</td>
<td>{ x2 y1 width height }</td>
</tr>
<tr>
<td>Vertically Moveable and Horizontally Moveable</td>
<td>#VMove, #HMove</td>
<td>{ x2 y2 width height }</td>
</tr>
<tr>
<td>Vertically Resizeable</td>
<td>#VResize</td>
<td>{ x1 y1 width y2 }</td>
</tr>
<tr>
<td>Horizontally Resizeable</td>
<td>#HResize</td>
<td>{ x1 y1 x2 height }</td>
</tr>
<tr>
<td>Vertically Resizeable and Horizontally Resizeable</td>
<td>#VResize, #HResize</td>
<td>{ x1 y1 x2 y2 }</td>
</tr>
</tbody>
</table>
Controller message protocol

As the user interacts with the system, the system sends messages to the controllers for each interface. These methods are implemented generically. As you work with ObjectStudio, you will probably subclass or write more specific versions of some of these methods.

This section explains how a controller accomplishes its work. It also provides a summary table of some commonly used methods.

System creates code. Recall that when you create an interface in the Designer, the system generates Smalltalk code that implements the interface. As part of this code, the system creates:

- A subclass of the class where you based the interface. (For example, if you based the interface on the Controller class, the system creates a subclass of Controller.)
- An instance of the new subclass.

Opening a controller. Recall that when the user opens an interface, the system displays the controller’s main form by default.

Initializing an interface

When the system loads an interface, it initiates these events:

1. The system sends the open message to the controller.
2. From the open method, the controller sends the open message to the main form.

About the main form. The main form has two roles in an interface:

- When the controller opens, the main form opens.
- When the main form closes, the controller closes.
Opening a form

When a form receives a request to open, the following events occur just before the form opens:

1. The form sends the openInitializationFor: message to its controller. The parameter is self, the form.
2. The controller determines whether the form being opened is the main form. If it is not, the default behavior is to do nothing.
3. If the form being opened is the main form, the controller sends the openInitialization message to itself.

**Subclassing:** Subclass one of these methods (openInitializationFor: or openInitialization) to do work before a form item opens. For example, you can subclass the openInitializationFor: method to initialize data, to set default options for form items, or to disable Buttons on the form.

When a form opens, it adds itself to the ActiveForms array.

After a form opens

Once a form opens and is visible, the following events occur:

1. The form sends the postOpenInitializationFor: message to its controller. The parameter is self, the form.
2. The controller determines whether the form that just opened is the main form. If it is not, the default behavior is to do nothing.
3. If the form that just opened is the main form, the controller sends the postOpenInitialization message to itself.

**Subclassing:** Subclass one of these methods to trigger actions when a form opens. For example, you can subclass the postOpenIntializationFor: method to open a form's subforms when the form opens.
Setting focus to a form

When a form receives focus, the form sends the activationFor: message to its controller. The parameter is self, the form.

The activationFor: method:

♦ Moves the form to the back of the ActiveForms array (the form is in the ActiveForms array when it opens)
♦ Redisplays selections
♦ Adds the window name to Window menus

Subclassing. Subclass this method (retain its default behavior) if you want the system to perform some action whenever the form receives focus. To do so, make the last line of the method a call to super, as follows:

```objective-c
activationFor: form
... super activationFor: form.
```

Closing a form

When the user closes a form by selecting Close from the System Menu, the form sends the closeRequestedBy: message to its controller. The method’s parameter is self, the form that is being closed.

Subclass the closeRequestedBy: method to perform clean-up tasks when a window closes. For example, you might want to subclass this method to close a database connection whenever a certain form closes.

Closing a form without using the system menu

You may need to write code to close a form by mechanisms other than the System Menu. For example, the form can include a Button that closes the form. Ordinarily, the code that you write to close a form sends the close message. However, the close method does not send the closeRequestedBy: message.

If you subclass the closeRequestedBy: method, you can ensure consistent behavior whenever the form closes if you send the closeRequested message instead of the close message. The closeRequested method sends the closeRequestedBy: message.
Method summary

The following table describes, in alphabetical order, some of the methods that the controller uses most commonly:

<table>
<thead>
<tr>
<th>Method</th>
<th>Called when</th>
</tr>
</thead>
<tbody>
<tr>
<td>activationFor:</td>
<td>A form receives focus.</td>
</tr>
<tr>
<td>closeRequestedBy:</td>
<td>A form closes.</td>
</tr>
<tr>
<td>finishInit</td>
<td>The initialize method finishes executing. (This is a class method.)</td>
</tr>
<tr>
<td>formSizeChangedFor:</td>
<td>A form is resized, maximized, or restored.</td>
</tr>
<tr>
<td>getColLabelsFor:from:to:</td>
<td>A form containing a Rectangle form item opens.</td>
</tr>
<tr>
<td>getRectValueFor:at:</td>
<td>A form containing a Rectangle form item opens.</td>
</tr>
<tr>
<td>getRowLabelsFor:From:To:</td>
<td>A form containing a Rectangle form item opens.</td>
</tr>
<tr>
<td>gotFocus:</td>
<td>A form item in the interface receives focus.</td>
</tr>
<tr>
<td>hasBeenClosed:</td>
<td>A form closes.</td>
</tr>
<tr>
<td>lostFocus:</td>
<td>A form item in the interface loses focus.</td>
</tr>
<tr>
<td>openInitialization</td>
<td>The main form is opening, just before it is displayed.</td>
</tr>
<tr>
<td>openInitializationFor:</td>
<td>The form is opening, just before it is displayed.</td>
</tr>
<tr>
<td>postOpenInitialization</td>
<td>The main form is opening, just after it is displayed.</td>
</tr>
<tr>
<td>postOpenInitializationFor:</td>
<td>A form is opening, just after it is displayed.</td>
</tr>
<tr>
<td>redrawDrawPad:</td>
<td>A form that contains a Draw Pad form item opens.</td>
</tr>
</tbody>
</table>
8. Responding to user interaction

Working with events

Once you define menu items and form items for an interface, you need to define methods that respond whenever a user interacts with the interface. User interactions, such as selecting a menu item or clicking a Button, are called events. After defining methods that respond to events, you associate the methods with the appropriate part of the interface.

The following table lists the types of events to which the system can respond:

<table>
<thead>
<tr>
<th>On a</th>
<th>You can define these event types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Accelerator key events</td>
</tr>
<tr>
<td></td>
<td>Return key events</td>
</tr>
<tr>
<td>Form item</td>
<td>Action events</td>
</tr>
<tr>
<td></td>
<td>Edit events</td>
</tr>
<tr>
<td></td>
<td>Hit key events</td>
</tr>
<tr>
<td>Form or form item</td>
<td>General events</td>
</tr>
</tbody>
</table>

The following sections describe:

♦ Methods you can write to respond to an event
♦ How to associate a method with the interface

This section assumes that you know how to use the Method Editor from within the Designer. If you need help with the Method Editor, see “Using the Method Editor to create a method” on page 61.

Form events

A form event is an event that can occur whenever the form has focus. For example, if you declare that the accelerator, CTRL+V, performs a Paste operation, the user can press accelerator keys when the form has focus. The form contains information about how to respond to the event.

The following table describes the types of events that can occur on a form that has focus:

<table>
<thead>
<tr>
<th>Event type</th>
<th>Occurs when the user</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator key events</td>
<td>Presses a predefined set of keys such as ALT+F or CTRL+C.</td>
</tr>
<tr>
<td>Return key events</td>
<td>Presses ENTER.</td>
</tr>
</tbody>
</table>
Defining accelerator key events

An accelerator key event occurs when the user presses a predefined combination of keys like CTRL+C or SHIFT+CTRL+A; the accelerator key event then executes a method.

**Menus:** Accelerators are often used in menus.

**Accelerators vs. mnemonics:** Because they both involve keyboard shortcuts, it can be easy to confuse accelerators and mnemonics. An accelerator is a keyboard shortcut, consisting of one key or a combination of keys, that executes a method. On the other hand, a mnemonic is a keyboard shortcut, indicated by an underlined letter, that is activated by pressing ALT+the underlined letter. For information on mnemonics, see “Creating mnemonics for form items” on page 139.

**Valid accelerator key combinations:** A valid accelerator key combination is represented to ObjectStudio as a Symbol such as #CtrlA, #CtrlAltNewLine, #ShiftF1, or just #A. It consists of a portion that is optional, which is one or more of the CTRL, ALT, and SHIFT keys, along with the required portion, which is one of the following:

- A number. 0-9.
- A function key. F1-F12.
- A keypad number key. KP_0-KP_9.
- A special identifier. See the following list:
  - #Backspace
  - #Break
  - #Delete
  - #Down
  - #End
  - #Esc
  - #Home
  - #Insert
  - #Left
  - #NewLine (Enter key)
  - #NumLock
  - #PageDown
  - #PageUp
  - #Pause
  - #Right
  - #Space
  - #Tab
  - #Up

**Testing your accelerator key combinations.** For various reasons, some key combinations will simply not work as expected. For example, Windows intercepts certain combinations, such as CTRL+Esc, preventing ObjectStudio from processing them. For this reason, you should test all accelerator key combinations in your applications and make sure these combinations behave as expected. If they do not, try different ones.
Assigning an accelerator key

To assign an accelerator key:

1. Select Form > Accelerator keys from the Designer menu.

   The Accelerator Keys dialog box displays, as shown here:

   ![Accelerator Keys dialog box]

   Note that the dialog box has a List Box containing the available accelerator keys.

2. Add a new accelerator key:
   A. Click New.
   
   The String dialog box displays.
   B. Type a new accelerator key name in the Enter new accelerator key name entry field (for example, AltC).
   C. Click OK.

3. Assign a method to an accelerator key:
   A. Select the accelerator key to which you want to assign a method.
   B. Click Assign.
   
   The Method Assignment dialog box displays.
   C. Select the method from the Available Methods list box.
   D. Click Copy to add the method to the Assigned Methods list box.
   E. Click OK.

4. Click Close to close the Accelerator Keys dialog box.
Deleting an accelerator key
To delete an accelerator key:
1. Select the accelerator key to delete from the Accelerator Keys dialog box.
2. Click Delete.
The system removes the accelerator key from the list.

Defining return key events
A return key event occurs when the user presses ENTER. The event applies to the form item that has focus. You can define more than one return key event for a form. When the user presses ENTER, the system selects the method to execute based on which form item has focus.

To define a return key method:
1. Create a form and form items.
2. Create the methods to assign to the ENTER key.
3. If the interface has more than one form, select the form to which you want to assign a return key method.
4. Select Form > Return key methods from the Designer menu.

The Return Key Method Assignment dialog box appears:

5. Select the form item or items to which you want to assign a return key method from the Items list box.
6. Select the method to assign as a return key method from the Available Methods list box.
7. Click Copy to add the method to the Assigned Methods list box.
Deleting a return key method
To delete a return key method:
1. Select a method to delete from the Assigned Methods list box.
2. Click Delete.
The system removes the method from the Assigned Methods list box.

Exiting the Return Key Method Assignment dialog box
To exit the dialog box, click OK to save your changes or click Cancel to exit without saving.

Form item events
A form item event is an event that occurs on a specific form item. For example, clicking a Button or selecting a List Box item are form item events.

Defining form item events
To define a form item event:
1. Create a form and form items.
2. Create a method for a form item event as described in the following table:

<table>
<thead>
<tr>
<th>Event</th>
<th>Create a method with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>No parameters. It occurs when the user interacts directly with a form item (for example, clicks a Button, types in an Entry Field, or selects from a menu).</td>
</tr>
<tr>
<td>Edit</td>
<td>No parameters or one parameter. If you supply a parameter, it represents the position of the list item that the user selected. For example, if the user selected the third item, the value of the parameter is 3. It occurs when the user selects an item from a List Box type of form item (either double-clicks the item or selects it and presses ENTER).</td>
</tr>
<tr>
<td>Hit key</td>
<td>No parameters or one parameter. If you supply a parameter, it represents a symbol corresponding to the name of the key that the user pressed. It occurs when the user presses a nonprinting key (such as INSERT or DELETE) when the form item has focus.</td>
</tr>
</tbody>
</table>

These methods can accept an optional parameter, which displays after any required parameters. The optional parameter represents the name of the form item to which the event happens.
3. Double-click the form item for which you want to create a form item event.
   The options dialog box for the form item displays.
4. Click **Methods** on the options dialog box.
   The Method Assignment dialog box displays, as shown here:

![Method Assignment dialog box](image)

Note that the **Item Name** text field contains the name of the form item to which you are assigning a method. There is also a list box of available methods.

If your Designer session is based on a subclass of a subclass of Controller or InterfaceComponent, you can check the Inherited Methods check box to list all methods defined in superclasses and in this class.
Assigning a method
To assign a method to the form item:
1. Select the Event Type radio button that corresponds to the type of event for which you are defining a method.
2. Select the method that you want to assign.
3. Click Copy.
The method displays in the Assigned Methods list box.

 Deleting an assigned method
To delete an assigned method:
1. From the Assigned Methods dialog box, select the method that you want to delete.
2. Click Delete.
The method is removed from the Assigned Methods dialog box.

Exiting the Method Assignment dialog box
To exit the dialog box, click OK to save your changes or click Cancel to exit without saving your changes.

Smalltalk details. The controller for each interface contains dictionaries that track each type of form item event. These dictionaries are:
- actionDict
- hitKeyDict
- editDict
The dictionaries store form item names and method names as symbols. To add new items or to retrieve existing items, use normal dictionary access methods, as described in the ObjectStudio Smalltalk User’s Guide, P40-3810.
General events

An event is any user interface change that results from user interaction. For example, an event occurs whenever the user:

♦ Presses or releases a mouse button
♦ Assigns focus to a new window
♦ Enters text in an Entry Field

When an event occurs, the system broadcasts a message to all objects.

Introducing the Event Editor

You use the Event Editor to specify how a specific object should respond when it receives notification that the event occurred. Typically, the object sends a message to cause a change in some part of the application.

Examples later in this section show how to use the Event Editor to:

♦ Change the font of the Employee list box whenever the user selects a new font from the font selector.
♦ Close a form when the user presses mouse button 1.

Definitions. The following table defines some terms that you need to understand as you work with the Event Editor:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>Object that triggers the event</td>
<td>If the user clicks a Button, the Button object is the sender of the event.</td>
</tr>
<tr>
<td>Receiver</td>
<td>Object that receives the broadcast message</td>
<td>If the clicked Button causes a List Box to be updated, the List Box is the receiver of the message.</td>
</tr>
<tr>
<td>Hooked</td>
<td>State of an object defined as a receiver of an event message</td>
<td>Once you use the Event Editor to define how the List Box responds to the clicked Button, the List Box is hooked to the clicked button event.</td>
</tr>
</tbody>
</table>
Opening the Event Editor
To display the Event Editor, select from the Designer menu either:

- **Controller > Events**, if you are creating an interface
- **Component > Events**, if you are creating an interface component

The Event Editor displays, as shown here:

![Event Editor](image)

**Overview.** You read and work with the elements in the Event Editor in a clockwise fashion, moving from the upper-left corner to the lower-left corner. In other words, define an event in this sequence:

1. Select a sender object.
2. Select an event name.
3. Select an event receiver and the message to send to it.
4. Click **Add** to add the message.

The system adds the event you define to the Hooked receiver objects and Messages list boxes.
The following table describes the elements of the Event Editor in more detail:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender objects</td>
<td>Lists all graphical objects that can generate an event. Select one sender object.</td>
</tr>
<tr>
<td>Event names</td>
<td>Lists all events supported by the sender you selected. Select an event.</td>
</tr>
<tr>
<td>Specify event receiver</td>
<td>Allows you to specify the messages to send when an event occurs. The box elements are:</td>
</tr>
<tr>
<td></td>
<td>† <strong>Receiver.</strong> Select the object that is the receiver of the message.</td>
</tr>
<tr>
<td></td>
<td>† <strong>Messages.</strong> Select the message to send to the receiver when the event occurs. Make sure that the selected message is not automatically sent through other processes. For example, the ‘open’ event will automatically send the postOpenInitialization message.</td>
</tr>
<tr>
<td></td>
<td>† <strong>Inherited.</strong> Check this box to display all messages, including the ones that the receiver object inherits.</td>
</tr>
<tr>
<td></td>
<td>† <strong>Method.</strong> Click this Button for convenient access to the Method Editor. (The system enables this Button only when you select an interface component or a controller as the event receiver.)</td>
</tr>
<tr>
<td>Messages and Hooked receiver objects</td>
<td>Synchronized lists that show all messages and receivers for each sender and event in the order in which they execute. To reorder, select one item and drag it to its new location.</td>
</tr>
<tr>
<td>Add button</td>
<td>Adds the message and its receiver to the Messages and Hooked receiver objects list boxes.</td>
</tr>
<tr>
<td>Remove button</td>
<td>Removes the selected message and object from the Messages and Hooked receiver objects list boxes.</td>
</tr>
<tr>
<td>OK button</td>
<td>Exits the dialog box. Click OK to save your changes.</td>
</tr>
<tr>
<td>Cancel button</td>
<td>Exits the dialog box. Click Cancel to exit without saving.</td>
</tr>
</tbody>
</table>
Completing the font selector

In this section, you will complete the font selector interface you created in “Using the font selector in an interface” on page 263. This section shows you how to use the Event Editor to change the font in the Employees list box whenever you select a new font.

To complete the font selector example:

1. On the Desktop, select the File > Load File menu option.
   The Open dialog box appears.
2. Use the Open dialog box to load fontsel.cls.
   fontsel.cls was the file created in “Saving and testing your work” on page 261.
3. Select the File > Load File menu option.
   The Open dialog box appears.
4. Use the Open dialog box to load fontinit.cls.
   fontinit.cls was the file saved in “Saving and testing the font selector” on page 265.
5. In the Designer, select the Controller > Events menu item.
   The Event Editor displays.
6. Select fontSelector1 (Font Selector) from the Sender objects list box.
   The system displays events for the sender in the Event names list box.
7. Select the changed: event from the Event names list box.
8. Select listbox1 (List box) in form Employees from the Specify Event Receiver list box.
9. Select the setFormatTo: message from the Message list box.
10. Click the **Add** button.

The system adds the receiver and message to the Hooked receiver objects and Messages list boxes. The Event Editor displays, as shown here:

11. Click **OK**.
Saving and testing the interface
To save and test the interface:

1. Select **File > Save** from the Designer menu.
2. Select **File > Test interface** from the Designer menu.
3. Select a new font from the drop-down list box.

If everything works as expected, the following happens, as shown in the following figure:

- The Entry Field displays the word Example in the font you selected
- The List Box displays its items in the font you selected

![Example interface](image)

When you have finished testing, return to the Designer by selecting **File > Edit interface** from the Designer menu.
Closing a form

In this section, you enhance the Employees interface so that when the user presses and releases mouse button 1 on the background of the main form, the form closes.

This example is for illustration purposes only. We recommend that you use more conventional techniques to close forms in your application.

To create the event:

1. Select Controller > Events from the Designer menu to display the Event Editor.
2. Select Employees (Form) from the Sender objects list box.
3. Select the wmButtonClick event from the Event names list box.
4. Select Employees (Form) from the Specify event receiver list box.
5. Select the close message from the Message list box.
6. Click Add.

The Event Editor displays, as shown here:

7. Click OK to save your changes.
Saving and testing the interface
To save and test the interface:

1. Select **File > Save** from the Designer menu.
2. Select **File > Test interface** from the Designer menu.
3. On the background of the Employees dialog box, press and release mouse button 1.

When you release mouse button 1, the Employees dialog box closes.

When you have finished testing, return to the Designer by selecting **File > Edit interface** from the Designer menu.

Using the Model View Controller approach

**Model View Controller (MVC)** is a mechanism for responding to user interaction. You also can use MVC to react when changes in one object affect other objects.

The MVC approach to design separates data and logic (together, called the model) from the interface (called the view). Benefits of using MVC are:

- The model is reusable. You can build different views that all use the same underlying model. For each view, you can display the model’s data differently.
- You can display the same model in different views simultaneously. When the model changes, the system updates all views.

**Example.** Consider a word processor that you use to create documents. You can view the same document in several modes:

- Final format
- Draft format
- With codes visible
- As an outline
Each mode provides a different view of the underlying model, the document, as shown in the following figure. When you change the model in one view, the system reflects the change in the other views.

VIEW #1
Visible Codes

This a document that can be viewed in various ways. One view is as (b) plain ASCII text with codes embedded. Another view is (b) formatted WYSIWYG. A third view is an (b) outline view. (b) Other views are also possible.

VIEW #2
WYSIWYG

This a document that can be viewed in various ways. One view is as plain ASCII text with codes embedded. Another view is formatted WYSIWYG. A third view is an outline view. Other views are also possible.

VIEW #3
Outline View

1. This a document
   1.a One view is as plain ASCII text
   1.b Another view is formatted
   1.c A third view is an outline view
   1.d Other views are also possible.

2. This a document
   2.a One view is as plain ASCII text
   2.b Another view is formatted
   2.c A third view is an outline view
   2.d Other views are also possible.

3. This a document
   3.a One view is as plain ASCII text
   3.b Another view is formatted
   3.c A third view is an outline view
   3.d Other views are also possible.

MODEL
A Word-Processing Document
How MVC works

MVC works via a set of declared links between application objects. These links are directional. When you declare an MVC link, you make one object depend on another object. The links ensure that when one object changes, all objects that depend on it are notified of the change. The dependent objects can then react appropriately (or not at all) and continue the cycle.

Using MVC in an application

There are two ways to take advantage of the MVC mechanism in an application:

♦ Use the Designer to create links. The system generates the necessary MVC code.

♦ Write code that implements MVC links. The code updates dependent objects when they are notified of a change.

The following sections further describe the two ways of using MVC.
Using the Designer

This section presents an example of how the Designer uses MVC to react to changes.

**MVC objects**

MVC divides objects of an application into three areas of responsibility, as described in the following table:

<table>
<thead>
<tr>
<th>Object type</th>
<th>MVC responsibility</th>
<th>Implemented by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Maintains the data that other application objects display.</td>
<td>An object of any class.</td>
</tr>
<tr>
<td>View</td>
<td>Visually displays the data maintained by the model.</td>
<td>An object of class ControllerItem.</td>
</tr>
<tr>
<td>Controller</td>
<td>Owns and controls the view.</td>
<td>An object of class Controller.</td>
</tr>
</tbody>
</table>

Recall that MVC separates the view (the presentation of your data) from the model (the data) to maintain independence between the two. This separation allows you to create many views of the same data.

For example, you can display employee salary as a string in an Entry Field, as a bar in a Bar Chart, and as items in a List Box simultaneously. As the following figure illustrates, the Model is the database row, the Controller is the interface controller, and the Views are the controller items that display salary:
Building the interface

This section describes how to build the MVC interface. It assumes that you are familiar with the Designer. If you need more detailed instructions, see “1. Introducing the Designer” on page 11, “2. Creating forms” on page 45, and “3. Creating form items” on page 120.

To build the interface described in “MVC objects” on page 327:

1. Select File > New > Interface to open the Designer.
2. Add a Bar Chart, a List Box, and an Entry Field to the main form.
3. For each item, link the item to its source data:
   A. Double-click the item to display the Form Options dialog box for the item.
   B. Click Link to display the Link Hookups dialog box.
   C. From left to right, select Table, Employee, and salary.
   D. Click OK to exit the Link Hookups dialog box.
   E. Click OK to exit the Form Options dialog box.

Modifying the Chart

Modify the Bar Chart:

1. Resize the Chart to at least double its original height and width.
2. Double-click the Chart to display the Form Chart Options dialog box.
3. Edit the Chart’s attributes:
   A. Click Options to display the Form Item Options dialog box.
   B. Check the Enable Mouse and Track Mouse check boxes.
   C. Click OK to close the dialog box.
4. Select the color for the Chart’s bars from the Color drop-down list box. For this example, select Blue.
5. Select the Chart’s selection color from the Selection color drop-down list box. For this example, select Red.
6. Click OK to close the dialog box.
Saving and testing the interface

To save and test the interface:

1. Select File > Save as to save the interface so that you can refer to it later.

2. Select File > Test interface from the Designer menu.

Note that:

♦ Each view displays the same value. This value reflects the data in the current database row. If you click another bar in the Chart, you change the current row.

♦ Each view updates itself to reflect the changed data.

Comments: This example demonstrates how the Designer works with MVC. When you link each form item to the Employee table, you declare that the item depends on the table. When you change the current database row, the system notifies the dependents of the change, and the dependents update themselves accordingly.

The example that is introduced in “Writing MVC code: Example overview” on page 344 presents a more detailed look at working with the Designer. For now, you should know that ObjectStudio contains a framework built on top of MVC. The Designer generates code that is at a higher level than the basic MVC code that you examine later in this section. However, the high-level code is constructed from MVC code.
Writing MVC code

This section describes the objects and methods you need when you write MVC code.

View and Controller objects. ObjectStudio implements the View and Controller areas of MVC by using objects of the classes:

- Controller
- ControllerItem (instances of this class are called cItems)
- Form
-FormItem (instances of this class are called fItems)

For a general introduction to these classes, see "Designer concepts" on page 13. The following figure illustrates the relationship between these items:
**Controller.** In ObjectStudio, the controller acts as an interface between the user and the rest of the system. The controller:

- Responds to user events
- Keeps track of its forms in the formDict dictionary (the forms, in turn, keep track of their form items)
- Keeps track of its controller items (cItems) in its cItemDict
- Tracks the links between view objects and model objects

**ControllerItem and FormItem:** For each item on a form, the system creates a corresponding instance of a ControllerItem and a FormItem. The FormItem instance manages how the item displays. The ControllerItem instance handles messages from the controller.
Model objects: links and dependencies

A model object can be of any class. The controller maintains links between model objects and view objects so that when a change occurs to an object, the controller can notify its dependents.

There are two types of links, as described in the following table:

<table>
<thead>
<tr>
<th>Link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>Changes in the model affect the view. The view object depends on the model object.</td>
</tr>
<tr>
<td>Update</td>
<td>Changes in the view (for example, when a user edits a value) affect the model. The model object depends on the view object.</td>
</tr>
</tbody>
</table>

A view object and a model object can have a regular link and an update link. In this case, the view object is a dependent of the model object and the model object is a dependent of the view object.

The following figure illustrates the different types of links:

In the figure, an arrow pointing away from an object towards another object indicates that the second object is dependent on the first.

How it works in Smalltalk: overview. When an object changes, it sends a changed message to itself. The default implementation of the changed method, in turn, sends an update message to all dependents of the changed object. Each dependent object must implement its own update method to process the change.
Link implementation

Link dictionaries. The controller maintains two dictionaries to manage links:

- The `clitemToLinks` dictionary, which contains regular links
- The `clitemToUpdateLinks` dictionary, which contains update links

These two dictionaries are defined as instance variables in class `Controller`. The following figure illustrates the two dictionaries. It also shows the relationships between the controller, the view objects (`clitems`), and the model objects (`Linkobjects`).

The link information is stored in the `clitemToLinks` and `clitemToUpdateLinks` dictionaries. The link information moves to the dependencyDictionary when the controller opens. The dependencyDictionary stores information about the links between the objects (for example, a database table and a stringCtrl).
**Link objects.** Each time you declare a link between two objects, the system creates an instance of a subclass of ObjectLink. The system provides built-in links, as described in the following table:

<table>
<thead>
<tr>
<th>Link to</th>
<th>Subclass of ObjectLink</th>
<th>Description of object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>TableLink</td>
<td>Represents a database table. It is an array of records, each of which is an array of field values.</td>
</tr>
<tr>
<td>SQLSelect</td>
<td>SQLLink</td>
<td>Represents the result of an SQL SELECT statement. It is a set of Tables and cursors.</td>
</tr>
<tr>
<td>Model</td>
<td>ModelLink</td>
<td>Represents the result of working with the Model Editor. It is a set of block variables and calculated expressions.</td>
</tr>
<tr>
<td>Controller</td>
<td>LinkCtrl</td>
<td>A controller that you create in the Designer.</td>
</tr>
</tbody>
</table>

Each of these subclasses implements behavior for specific types of links. For example, the Designer uses TableLink to link a form item to a database table.

The controller contains static definitions of its links. When the controller’s class file is loaded into the ObjectStudio environment, the link objects are created and placed into the appropriate link dictionary.
Links during application execution

This section describes how the system handles link objects while the application runs.

When the controller loads into memory, the initialize method sends the updateDependencies message to the controller. The updateDependencies method traverses each entry in the links dictionaries (cItemToLinks and cItemToUpdateLinks). For each entry, it creates dynamic dependencies between the controller item and the model object.

Regular links: For a regular link, recall that a model can be associated with more than one view, but the source data for a view can only come from one model. Therefore, as the following figure illustrates, each controller item (view object), will depend on at most one model object. Each model object, however, can have links to multiple controller items.
Here is the sequence of events:

1. The system sends the `getObject` message to each link object for each controller item.
2. The `getObject` method finds the model object on which the controller item depends.
3. The method adds the controller item to the model object’s dependency dictionary to establish a run-time link.

**Update links.** For an update link, when a controller item changes, more than one model object can be affected. Therefore, as the following figure illustrates, each controller item can have links to multiple model objects:

Here is the sequence of events:

1. The system sends the `getObject` message to each link object for each controller item.
2. The `getObject` method finds the model object that depends on the controller item.
3. The method adds the model object to the controller item’s dependency dictionary to establish a run-time link.
Creating dependencies

The following table describes methods that declare dependencies between objects:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addDependent:</td>
<td>Declares that the object in the parameter depends on the receiver object.</td>
</tr>
<tr>
<td>addDependent:field:</td>
<td>Declares that the object in parameter 1 depends on the field in parameter 2, which is part of the receiver object.</td>
</tr>
</tbody>
</table>

**Example.** To declare that aPersonView depends on the first name field of aPerson, type:

```plaintext
aPerson addDependent: aPersonView
    field: #firstName.
```
Removing dependencies

The following table describes methods that remove dependencies between objects:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeDependent:</td>
<td>Removes the object in the parameter from the dependency dictionary that belongs to the receiver.</td>
</tr>
<tr>
<td>removeSelfFromDependents</td>
<td>Removes the receiver from all dependency lists. Implement this method by making the receiver of this message the parameter to removeDependent:.</td>
</tr>
<tr>
<td>removeDependent:field:</td>
<td>Removes the dependency from the object in parameter 1 to the field in parameter 2. The field is part of the receiver.</td>
</tr>
<tr>
<td>release</td>
<td>Informs the receiver that an object that the receiver depends on is closed.</td>
</tr>
</tbody>
</table>

Examples. To remove a dependency between aPersonView and the firstName field of aPerson, type:

```objective-c
aPerson removeDependent: aPersonView
    field: #firstName.
```

To remove all dependencies from aPersonView, type:

```objective-c
aPersonView removeSelfFromDependents.
```

Getting information about dependencies

The following table describes methods that retrieve information about dependencies:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependents</td>
<td>Returns a set of all objects that depend directly on the receiver.</td>
</tr>
<tr>
<td>allDependents</td>
<td>Returns a set of all objects that directly or indirectly depend on the receiver.</td>
</tr>
<tr>
<td>hasDependent</td>
<td>Returns true if there is an object that depends on the receiver. It returns false otherwise.</td>
</tr>
</tbody>
</table>
Notifying dependents of a change
When an object changes, the following events occur:
1. The object sends the changed message to itself.
2. The changed method iterates through the object’s dependencyDictionary.
3. The object sends an update message to each dependent that it wants to notify.
4. Each dependent executes its update method to process the change. (You must implement a customized update method for each dependent.)

broadcast methods: The broadcast and broadcast:with methods send messages to all dependents. These methods can be used to notify dependents of a change, as described in the previous discussion.
changed methods: The parameters to the changed methods can contain any type of information necessary to the application. Ensure that when you send a changed message, the receiver object has implemented the changed method appropriately.

The following table describes methods that are variations of the changed method:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changed</td>
<td>Sends the update: message to inform dependents that the receiver changed.</td>
</tr>
<tr>
<td>changed:</td>
<td>Sends the update: message to inform dependents that the part of the receiver, specified by the parameter, changed.</td>
</tr>
</tbody>
</table>
| changed:type: | Used by controller items to indicate a change by sending the message update:with:with:with: to inform dependents that an aspect of the receiver changed. The first parameter is nil or the index at which the change occurred. The second parameter indicates the type of change. It can be either:  
  ♦ #Position, which informs dependents of a location change  
  ♦ #Value, which informs dependents of a value change |
| changed:with: | Sends the update:with: message to inform dependents that an aspect of the receiver, specified by the first parameter, changed. The second parameter provides additional information about the change. |
| changed:with:with: | Sends the update:with:with: message to inform dependents that an aspect of the receiver, specified by the first parameter, changed. The second and third parameters provide additional information about the change. |
**update methods:** The changed methods listed in the previous paragraphs send an update message to the object’s dependents. The update message has the following variations:

- update:
- update:with:
- update:with:with:
- update:with:with:with:

Parameters to the update methods can contain any type of information. Ensure that when you send an update message, the receiver object has implemented the update method appropriately.

When the changed:type: method sends the message update:with:with:with:, the parameters to the update method are:

- Parameter 1 is the object that changed.
- Parameter 2 is the linked field name.
- Parameter 3 is nil or the index value.
- Parameter 4 is either #Position or #Value.
Example 1. To notify a model object that a controller item has changed, send the following message:

```
update: object with: field with: position
    with: type
```

The following table describes the parameters in the previous code:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Name of the changed object.</td>
</tr>
<tr>
<td>field</td>
<td>The field that was changed in the object.</td>
</tr>
<tr>
<td>position</td>
<td>The position in the object that was changed.</td>
</tr>
<tr>
<td>Type</td>
<td>The type of change that occurred.</td>
</tr>
</tbody>
</table>

Example 2. In this example, consider an Entry Field with an instance variable, employeeName. The Entry Field depends on a database table. The following points explain how the system establishes and maintains the dependency link:

- To declare the dependency, the Entry Field sends the following message to the database table:

```
addDependent: employeeName field: #NAME.
```

- Whenever the table receives the setPosition: message, it sends the following message to the Entry Field:

```
update: employeeName with: #NAME
    with: tablePosition with: #Position.
```

The following table describes the parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeName</td>
<td>The database table on which the Entry Field depends.</td>
</tr>
<tr>
<td>#NAME</td>
<td>The name of the table column on which the Entry Field depends.</td>
</tr>
<tr>
<td>tablePosition</td>
<td>The current position in the table.</td>
</tr>
<tr>
<td>#Position</td>
<td>The type of change that occurred.</td>
</tr>
</tbody>
</table>
Working with the Designer

The following table describes methods that you use when you create dependencies in the Designer:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register:</td>
<td>Creates a link to the object. The receiver is the ClassLink class, which manages the classes that you can link to from the Designer</td>
</tr>
<tr>
<td>getLinkAttributesFor:</td>
<td>Gets a list of the fields in the parameter that can be linked to. The return value is a dictionary with field name as keys and field types as values.</td>
</tr>
<tr>
<td>getLinkInstancesFor:</td>
<td>Gets a list of instances for the parameter that can be linked to. The return value is an array of symbols representing the available instances.</td>
</tr>
<tr>
<td>getNamed:</td>
<td>Retrieves the appropriate instance of a link. The parameter is the instance name. The return value is the actual instance.</td>
</tr>
</tbody>
</table>

For examples of how to use these methods, see the example in “Writing MVC code: Example overview” on page 344.
Writing MVC code: Example overview

This section describes an extended example that is centered around two classes:

♦ Instances of the Account class that have an account number and a current balance

♦ Instances of the Transaction class that have an account number and the amount to add or subtract from the account

The example is divided into three parts:

♦ In “MVC example: Part 1” on page 345, you create the two classes and declare that each instance of Account depends on instances of Transaction.

♦ In “MVC example: Part 2” on page 351, you create a form item in the Designer and declare a regular link to instances of Account.

♦ In “MVC example: Part 3” on page 354, you create another form item in the Designer and declare an update link to instances of Transaction.

The examples are lengthy and can take up to an hour to work through. However, once you understand how they work, you can write MVC code for your own application.
MVC example: Part 1

In this part of the example, you:
♦ Create the Account class.
♦ Create the Transaction class.
♦ Test your work.

Creating the Account class

Create the Account class with the properties:

<table>
<thead>
<tr>
<th>Class Name:</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superclass:</td>
<td>Object</td>
</tr>
</tbody>
</table>
| Instance variables: | accountNumber  
                     | balance    |
| Class variable:   | accountDict |
| Save to file:     | account.cls |

where:
- the accountDict class variable is a dictionary that keeps track of instances of Account
- the keys are account numbers
- the values are the instances
Class methods. Create the following class methods. Remember to select the Class radio button before starting.

initialize

"initializes the accountDict class variable"

accountDict := IdentityDictionary new.

new: acct

"Creates a new instance of Account. Adds it to the accountDict."

| obj |
obj := self new.
obj accountNumber: acct.
obj balance: 0.
accountDict at: (acct asSymbol) put: obj.
^ obj.

lookUp: obj

"Called whenever an instance of Transaction is created. The method makes an instance of Account depend on the parameter, which is the new transaction."

obj addDependent:

(accountDict at: (obj accountNumber)).

Note that the lookUp: method sends the addDependent: message.
**Instance methods.** Create the following instance methods. Remember to select the Instance radio button before starting. The following methods are set and get methods:

- `accountNumber`
  
  `^ accountNumber.`

- `accountNumber: acct`
  
  `accountNumber := (acct asSymbol).`

- `balance`
  
  `^ balance.`

- `balance: amt`
  
  `balance := amt.`

The following method corresponds to a changed: method in the Transaction object:

- `update: amount`
  
  "Updates the Account balance in response to a change in a Transaction."

  `balance := balance + amount.`
Creating the Transaction class

Create the Transaction class with the properties:

<table>
<thead>
<tr>
<th>Class Name:</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superclass:</td>
<td>Object</td>
</tr>
<tr>
<td>Instance variables:</td>
<td>transNumber, accountNumber, amount</td>
</tr>
<tr>
<td>Class variables:</td>
<td>nextTransNumber, transactionDict</td>
</tr>
<tr>
<td>Save to file:</td>
<td>transact.cls</td>
</tr>
</tbody>
</table>

where:

- the `nextTransNumber` class variable contains the number that will be assigned to the next transaction.
- the `transactionDict` class variable is a dictionary that keeps track of instances of Transaction. The keys are transaction numbers and the values are the instances.

Class methods. Create the following class methods. Remember to select the Class radio button before starting.

initialize

```
nextTransNumber := 1.
transactionDict := IdentityDictionary new.
```

new: acct

"Creates an instance of Transaction. Sends the lookUp message to the Account class and passes the new instance as the parameter."

```
| obj |
obj := self new.
obj transNumber: nextTransNumber.
obj accountNumber: acct.
obj amount: 0.
transactionDict at: (nextTransNumber asSymbol) put: obj.
nextTransNumber := nextTransNumber + 1.
Account lookUp: obj.
^ obj.
"
**Instance methods.** Create the following instance methods. Remember to select the Instance radio button before starting.

The following methods are set and get methods:

- **accountNumber**
  
  ^ accountNumber.

- **accountNumber: acct**
  
  accountNumber := (acct asSymbol).

- **amount**
  
  ^ amount.

- **amount: value**
  
  "Sets the value of the transaction and calls the changed: method."
  
  amount := value.
  
  self changed: amount.

- **transNumber**
  
  ^ transNumber.

- **transNumber: val**
  
  transNumber := (val asSymbol).
Testing your work

Open the System Transcript and the Workspace. Type and execute the following in the Workspace:

```plaintext
Acct1 := Account new: #1234.
Acct2 := Account new: #9876.

Tran1 := Transaction new: #1234.
Tran2 := Transaction new: #1234.
Tran3 := Transaction new: #9876.

Tran1 amount: 25000.
Tran2 amount: -10000.
Tran3 amount: 1000.

Acct1 balance out.
Acct2 balance out.
```

If the test works successfully, the System Transcript window will display 15000 and 1000.

**What happens.** In this example, you create two accounts and three transactions. Message flow occurs as follows:

1. You create an instance of Transaction by calling the new: message.
2. The new: method sends the lookup: message to Account.
3. The lookup: method makes the account corresponding to the transaction depend on the transaction.
4. You assigned amounts to each transaction by sending the amount: message.
5. The amount: method sends the changed: message to self (Transaction).
6. As defined in Object (Transaction’s superclass), the changed: method sends the update: message to inform dependent objects of the change.
7. The update: method in Account adds the new amount to the balance in the account.
MVC example: Part 2

In this part of the example:

♦ Add code to the Account class
♦ Create an interface in the Designer, including a link to Account
♦ Test your work

Adding code to Account

In this section, you add code to the Account class.

Class methods. Modify the initialize method by adding:

ClassLink register: self.

Recall that when an object sends the register: message, the Designer allows you to create a link to the object.

Add the following class methods to Account. Remember to select the Class radio button before starting.

getLinkAttributesFor: instanceName

"Get linkable fields for the parameter. Return dictionary with the field name as the key and the field type as the value."

| dict |
dict := IdentityDictionary new.
dict at: #balance put: #Number.
^ dict.

getLinkInstancesFor: db

"Get linkable instances of the Account class with active database object as the parameter. Return an array representing available instances."

^ accountDict keysAsArray sort.

getNamed: key

"Retrieve the appropriate instance of a link. The parameter is the instance name."

^ (accountDict at: key).
**Instance methods.** Create the following instance method. Remember to select the Instance radio button before starting.

```smalltalk
at: fld

"Sets current values for Entry Fields and Sliders. Sets the selection pointer to a matching value for List Boxes."

^ self varAt: (fld asSymbol).
```

Add the following line to the update: and balance: methods:

```smalltalk
self changed: nil type: #Value.
```

**Reload the class.** Load the Account class. Remember to check the Reload check box in the Open dialog box.

**Create instances.** Create two instances of Account by typing and executing the following code in the Workspace:

```smalltalk
Account new: #1111.
Account new: #9999.
```

**Working with the Designer**

The changes listed previously allow you to link Entry Fields, List Boxes, and Sliders to instances of Account. Now create the following interface in the Designer:

1. Place one Number Field on an interface.
2. Display the Link Hookups dialog box (link the form item).
   
   The Link Hookups dialog box displays Account in the left column because you sent the register: message to ClassLink.
3. From left to right, select Account, 9999, balance.
   
   The Designer displays 9999 because it sent the getLinkInstancesFor: message to Account. It displays balance because it sent the getLinkAttributesFor: message to Account.
4. Save the interface (you will use it again in “MVC example: Part 3” on page 354).
Testing your work

To test your work:

1. Select File > Test interface from the Designer menu.

2. Type and execute the following code in the Workspace:

   ```
   T := Transaction new: #9999.
   T amount: 1234.56.
   ```

   If the test works successfully, the Number Field will display the amount of the transaction, 1234.56.

What happens. In this example, message flow occurs as follows:

1. You update the Transaction by sending the amount: message.

2. The amount: method sends the changed: message to self (Transaction).

3. The changed: method sends the update: message to inform dependent objects of the change.

4. The update: method in Account adds the new amount to the balance in the account. It also sends the changed:type: message to itself.

5. The changed:type: method sends the message to its dependents, in this case, the Number Field controller item.

6. The implementation of the update:with:with:with: method in the Number Field updates the value and redispalyes the Number Field.
MVC example: Part 3

In this part of the example:

♦ Add code to the Transaction class
♦ Use the Designer to add a form item to the interface and to create an update link to Transaction
♦ Test your work

Adding code to the Transaction class

In this section, you add code to the Transaction class. Some of the following code is nearly identical to the code you added to the Account class. In those cases, the explanations are minimal.

Class methods. Add the following line of code to the initialize method:

```
ClassLink register: self.
```

Add the following class methods to Transaction. Remember to select the Class radio button before starting.

```
getLinkInstancesFor: db
   ^ transactionDict keysAsArray sort.

getLinkAttributesFor: instanceName
   | dict |
   dict := IdentityDictionary new.
   dict at: #amount put: #Number.
   ^ dict.

getNamed: key
   ^ (transactionDict at: key).
```
**Instance method.** In this example, you create an update link from a controller item to an instance of Transaction. In other words, you make the instance of Transaction depend on the controller item. Recall that the dependent object must respond to the update:with:with:with: message. Add the following instance method:

```
update: obj with: fld with: pos with: type
  ((fld == #amount) & (type == #Value)) ifTrue:
    [ self amount: (obj getValue). ].
```

**Reload the class.** Load the Transaction class. Remember to check the Reload check box.

**Create an instance.** Display the System Transcript window and the Workspace. Type and execute the following code in the Workspace:

```
T := Transaction new: #9999.
T transNumber out.
```

Note the transaction number that displays in the System Transcript window; you will use it in the next few steps.
Working with the Designer

Use the Designer to edit the interface that you worked on in “MVC example: Part 2” on page 351 and then:

1. Add another Number Field to the interface.
2. Display the Link Hookups dialog box (link the form item).
   The Link Hookups dialog box should display Transaction in the left column.
3. From left to right, select Transaction, the transaction number that you noted earlier, and amount.
4. Save the interface.

Testing your work

To test your work:

1. Select File > Test interface from the Designer menu.
2. In the new Number Field, type a number and press ENTER.

If the test works successfully, the number you typed will display in the Account number field.

What happens. In this example, message flow occurs as follows:

1. Type a new number in the Transaction number field.
2. Number Field sends the changed:type: message to itself.
3. The changed:type: method sends the update:with:with:with: message to the dependents of Number Field, which is, in this case, the instance of Transaction.
4. The update:with:with:with: method sets the amount by sending the amount: message.
5. The cycle of events described at the end of “MVC example: Part 2” on page 351 occurs, resulting in an update to the Account number field.
Validating data

ObjectStudio allows you to add methods that validate data that the user enters. Validation occurs when the form item that contains the data loses focus. This section explains validation.

Menus, toolbars, scroll bars, and the status line do not get focus. Clicking on these items does not cause an item with focus to lose that focus.

Overview of the process

When a form item loses focus, the controller sends the lostFocus message to the form item. The lostFocus method initiates the two-step validation process:

1. Check whether the entry is valid.

2. React to the error if the entry is not valid.

Usually, you validate Entry Fields, Number Fields, and Date Fields. The parent form item class of these items is FormString. The corresponding parent controller item class of these items is StringCtrl.

Other types of items inherit their validation behavior from the FormItem class.
Default validation

ObjectStudio defines a validation chain of messages and methods. “Validating data at the controller level” on page 361 and “Validating data at the controller item level” on page 362 show you how to override the validation process at the controller level and at the controller item level, respectively.

The default events that occur when an object loses focus are:

1. The controller sends the lostFocus message to the form item.
2. The lostFocus method sends the lostFocus: message to the form that contains the form item. The parameter to lostFocus: is self.
3. The form sends the lostFocus: message to its controller. The parameter to this lostFocus: message is the instance of the controller item that corresponds to the form item that lost focus.
4. By default, lostFocus: as defined in the Controller class does nothing. It is designed to be subclassed.

Summary. The following table summarizes the events that occur when an item loses focus:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Sender</th>
<th>Message</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Controller</td>
<td>lostFocus</td>
<td>Form item that lost focus.</td>
</tr>
<tr>
<td>2</td>
<td>Form item</td>
<td>lostFocus: self</td>
<td>Form to which the form item belongs.</td>
</tr>
<tr>
<td>3</td>
<td>Form</td>
<td>lostFocus: cItem</td>
<td>Controller.</td>
</tr>
</tbody>
</table>
Validating data in FormString items

This section describes the default validation behavior for items of class FormString.

**Step 1—Validate data.** The following steps describe what happens after a FormString item loses focus:

1. The controller sends the lostFocus message to the FormString item.
2. The lostFocus method sends the performValidation message to the controller item that corresponds to the FormString item.
3. The controller item sends the performValidationFor: message to the controller. The parameter is self (the controller item).
4. The controller sends the validate message to the controller item.
5. The validate method validates the data and returns a value of true (data is valid) or false (data is not valid). The return value of the validate method is passed back to the lostFocus method.

The default implementation of the validate method, defined in class ControllerItem, is to return true. Therefore, ensure that the validate method is implemented for the controller item.

**Summary.** The following table summarizes the events that occur when a FormString item loses focus and the system validates the item’s data:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Sender</th>
<th>Message</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Controller</td>
<td>lostFocus</td>
<td>FormString (form item) that lost focus.</td>
</tr>
<tr>
<td>2</td>
<td>Form item</td>
<td>performValidation</td>
<td>Controller item corresponding to the form item.</td>
</tr>
<tr>
<td>3</td>
<td>Controller item</td>
<td>performValidationFor: self</td>
<td>Controller.</td>
</tr>
<tr>
<td>4</td>
<td>Controller</td>
<td>validate</td>
<td>Controller item.</td>
</tr>
</tbody>
</table>
Step 2—React to errors. If the validate method returns false to the lostFocus method, the system must react to the error.

The following steps describe the system’s default behavior:

1. The FormString item’s lostFocus method sends the invalidData message to its controller item.
2. The invalidData message sends the invalidDataFor: message to the controller. The parameter is self (the controller item).
3. By default, the invalidDataFor: method causes the system to beep.

Summary. The following table summarizes the events that occur when a FormString item contains invalid data:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Sender</th>
<th>Message</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Form item</td>
<td>invalidData</td>
<td>Controller item.</td>
</tr>
<tr>
<td>2</td>
<td>Controller item</td>
<td>invalidDataFor:</td>
<td>Controller.</td>
</tr>
<tr>
<td>3</td>
<td>Controller</td>
<td>A beep message</td>
<td>The system.</td>
</tr>
</tbody>
</table>
Validating data at the controller level

This section shows you how to override the default validation process at the controller level. It uses the example of an Entry Field that needs validation.

**Step 1—Validate data.** In the first step of the validation process, the system determines whether the data is valid. In Smalltalk terms, the goal of this step is to return a true (data is valid) or false (data is not valid) result to the lostFocus method.

To customize validation at the controller level, override the performValidationFor: method for the controller that owns the Entry Field. Then, ensure that performValidationFor:

- Performs the actual validation, or, through message sending, causes the validation to occur
- Returns either true (the data is valid) or false
- Calls the Controller class’ version of the method (send performValidationFor: to super) to ensure that the method continues to work for controller items that do not require validation

**Step 2—React to errors.** The goal of the second step is to react to an invalid data entry. By default, the system beeps. This section shows you how to give the user more, or different, feedback.

To customize reaction to invalid data at the controller level, subclass the invalidDataFor: method for the controller that owns the Entry Field. Then, ensure that invalidDataFor:

- Reacts appropriately to the error. For example, you may want to display a message box informing the user of the error.
- Calls the Controller class’ version of the method (send invalidDataFor: to super) to ensure that the method continues to work for controller items that do not require validation.
Validating data at the controller item level

This section describes how to override the default validation process at the controller item level. The following table compares the effects of overriding behavior at the controller level and at the controller item level:

<table>
<thead>
<tr>
<th>Overriding validation</th>
<th>Behavior</th>
<th>Reusability</th>
</tr>
</thead>
<tbody>
<tr>
<td>At the controller level</td>
<td>Specific to an interface</td>
<td>Unless another interface is a subclass of the current interface, you cannot easily reuse the behavior.</td>
</tr>
<tr>
<td>At the controller item level</td>
<td>Specific to a controller item</td>
<td>Once you add behavior to a controller item, you can reuse the behavior wherever you use the controller item.</td>
</tr>
</tbody>
</table>

At the controller item level, you can override default system behavior at Step 1 but not at Step 2 (see “Validating data at the controller item level” on page 362). You cannot override at Step 2, because the system handles reaction to errors at the controller level.

**Step 1—Validate data.** To customize validation at the controller item level, subclass validate. Ensure that validate:

- Performs the actual validation
- Returns a value of true (data is valid) or false

**Example.** The following code is the default implementation of validate in the DateCtrl class:

```plaintext
define: Validates data.
define: Validates data.
validate
  | val isValid |
  (val := self getValue) notNil
    ifTrue: [ val isDate. ].
  ^ true.
```


9. Extended interface example

Introduction

This chapter contains an extended example that shows how to build an ObjectStudio application that displays and edits a database table.

This chapter is designed so you can learn techniques by doing the examples. You may want to read the code that results from doing the examples here. Finished code for the interface is in the file ObjectStudio/tutorial/tableex.cls.

Overview of the example

In the example, you use the Designer’s linking facility to connect the interface and the database table. The example demonstrates:

♦ Creating a complex user interface that includes menus
♦ Synchronizing interface controls
♦ Opening and working with a subform
♦ Assigning methods to controls
♦ Working with a database table: sorting the data and adding, changing, and deleting records
The interface you build has two forms:

♦ **Main form.** The main form, as shown in the following figure, displays the database table and allows you to sort it according to several criteria. The three Buttons at the bottom of the form allow you to add, change, or delete a database record.

![Main form](image)

♦ **Employee dialog box.** The Employee dialog box, as shown here, provides a form for adding or changing a database record:

![Employee dialog box](image)
As you work through this chapter, you construct the interface in the following stages:

1. **Build the user interface.**
2. **Sort the table.**
3. **Synchronize radio buttons and menus.**
4. **Implement the Delete option.**
5. **Display the Employee dialog box.**
6. **Accept and save changes.**
7. **Implement the other Buttons.**

**Building the user interface**

In this section, you use the Designer to construct the user interface for the application. This section assumes that you are familiar with using the Designer and provides detailed instructions only for complex tasks. If there is a step that you do not know how to complete, refer to earlier chapters.

To start, open the Designer and display the New Item dialog box.

**Building the main form**

The following figure shows the main form for the interface. Use this figure as a guide to help you size and place controls on the form.

**Changing the form title**

When you open the Designer to create a new interface, the Designer creates the main form for you. Change the main form’s title to Table Example.
Creating the controls

In this section, you create and place the controls on the main form:

♦ **Static Text.** Create a Static Text item whose text is Employees.

♦ **Tabular List Box.** The Tabular List Box displays the list of employees. Create it and then:

  1. Assign the name tableBox to the Tabular List Box form item.
  2. On the Form Item Options dialog box, enable a horizontal scroll bar.
  3. Link the Tabular List Box to the Employee table. On the Link Hookups dialog box, select Table, Employee, and click OK.

♦ **Radio Buttons.** The Radio Buttons specify the criteria for sorting the database table. Create a group of Radio Buttons and then:

  1. Assign the name sortButtons to the radio button form item.
  2. Create Radio Buttons with labels and names exactly as shown in the following table. (To change the values, double-click the value name.) The # (pound sign) is visible only when you edit the value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>#name</td>
</tr>
<tr>
<td>Age</td>
<td>#age</td>
</tr>
<tr>
<td>City</td>
<td>#city</td>
</tr>
<tr>
<td>Salary</td>
<td>#salary</td>
</tr>
</tbody>
</table>

The values differ from the names so that the values exactly match fields in the database table, which allows you to do shortcut programming later.

♦ **Check Box.** The Check Box determines whether to sort the database table in ascending or descending order. Create a Check Box with name, descending, and text, Descending.

When the application runs and the Check Box is checked, the system sorts in descending order. When it is unchecked, the system sorts in ascending order.
Buttons. The Buttons control the operations that you can perform on database records: adding, changing, and deleting. Create Buttons with attributes as shown in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>addButton</td>
<td>Add</td>
</tr>
<tr>
<td>changeButton</td>
<td>Change</td>
</tr>
<tr>
<td>deleteButton</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Creating menus
The interface has two menus on the menu bar and one cascading menu:

- The Actions menu performs the same functions as the Add, Change, and Delete buttons.
- The Sort menu performs the same functions as the Sort radio buttons.
- The Order menu, a cascading menu, performs the same functions as the Descending check box.

As with the Radio Buttons, you create a value for each menu item that differs from the item's label. By creating a value for each menu item, you can take advantage of programming shortcuts later in the example.

Select Tools > Menu from the Designer menu to open the Menu editor. For more information about creating menus, see “Menu editor” on page 75.

Actions menu. To create the Actions menu:

1. Type Actions in the Caption entry field. You can fill in the other item settings if you want.
2. Type the captions and values described in the following table:

<table>
<thead>
<tr>
<th>Caption</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>#actionAdd</td>
</tr>
<tr>
<td>Change</td>
<td>#actionChange</td>
</tr>
<tr>
<td>Delete</td>
<td>#actionDelete</td>
</tr>
</tbody>
</table>

Remember to demote each menu item so that each item displays in the Actions menu.
Sort menu. To create the Sort menu:

1. Type Sort in the Caption entry field. You can fill in the other item settings if you want.

2. Select the Checked check box in the Options group box of the Menu Setup dialog box. This feature allows you to duplicate the functionality of the Sort radio buttons.

3. Click Add.

4. Type the captions and values described in the following table:

<table>
<thead>
<tr>
<th>Caption</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>#name</td>
</tr>
<tr>
<td>Age</td>
<td>#age</td>
</tr>
<tr>
<td>City</td>
<td>#city</td>
</tr>
<tr>
<td>Salary</td>
<td>#salary</td>
</tr>
<tr>
<td>– (separator)</td>
<td>– (default value)</td>
</tr>
<tr>
<td>Order</td>
<td>Order (default value)</td>
</tr>
</tbody>
</table>

Remember to demote each menu item so that each item displays in the Sort menu.

Order cascading menu. The Order menu duplicates the function of the Descending check box. It is a cascading menu attached to the Sort menu.

To create the Order cascading menu:

1. Select Order from the Menu Definition list box.

2. Click Clear to clear the Menu Item Settings entry fields.

3. Add the following menu items and values to the Order menu:

<table>
<thead>
<tr>
<th>Caption</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascending</td>
<td>#asc</td>
</tr>
<tr>
<td>Descending</td>
<td>#desc</td>
</tr>
</tbody>
</table>

Remember to demote each menu item so that each item displays in the Order cascading menu.

Cascading menus cannot have check marks.
Building the Employee dialog box

The following figure shows the Employee dialog box, a subform of the main form. Use this figure as a guide to help you size and place controls on the form.

The dialog box provides a form for adding or changing employee records. The dialog box is modal, which means that the user cannot interact with the main form until the dialog box is closed.

Creating the dialog box

To create the dialog box:

1. Select **Form > New modal dialog box**.
2. Select **Form > Change form** and in the Form Name entry field type: EmployeeDialog
3. Change the **Form Title** to Employee.
4. In the **Options** group box, ensure that the **Dialog Border** check box is checked.
Creating the controls

In this section, you create and place controls on the Employee dialog box.

Control names. Each control in a controller must have a unique name. When the controller contains only one form, it is easy to create unique names. When the controller has more than one form, you can run into problems with duplicate names.

For example, if you have two dialog boxes, each with an OK button, the Buttons must have different names. Of course, the text of each control can duplicate text in another control.

A simple solution is to prefix each control name with the name of its form. As you create controls for the Employee dialog box, use this naming strategy. This approach may seem cumbersome, especially if there is only one secondary form (as in this interface), but it pays for itself later when you either add a new form or maintain controls in an existing form.

You can devise your own solution—perhaps using prefixes or abbreviations for the forms—but it is important that you create a standard and use it consistently.

Static Text. Create the Static Text items:

♦ Name:
♦ Age:
♦ Sex:
♦ City:
♦ Salary:
Entry Fields. Create two Entry Fields with the names:

- name
- city

Number Fields. Create two Number Fields with the names:

- age
- salary

Drop-down List Box. Create a Drop-down List Box and then:

1. Assign it the name sex
2. Assign it two labels: male and female

Check Box. Create a Check Box and then:

1. Assign it the name married
2. Assign it the text Married

Later, you write code to convert the true or false value of the Check Box to the married or single value in the database table.

Add and Close buttons. The Add button commits an addition or change to the database table. The Close button closes the Employee dialog box. Create two Buttons and assign a name and text, as shown in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeDialogAdd</td>
<td>Add</td>
</tr>
<tr>
<td>employeeDialogClose</td>
<td>Close</td>
</tr>
</tbody>
</table>

Previous and Next buttons. The Previous and Next buttons change the selected record in the database table. Create two Buttons and assign a name and text, as shown in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmployeeDialogPrevious</td>
<td>&lt;&lt; Previous</td>
</tr>
<tr>
<td>employeeDialogNext</td>
<td>Next &gt;&gt;</td>
</tr>
</tbody>
</table>
 Initializing forms

By default, the system provides the openInitialization method, which executes when the controller opens, and initializes the main form. However, the controller that you are building has two forms, and you want to initialize each form when it opens. For this purpose, you can implement the openInitializationFor: method.

This method takes one parameter, the form that is opening. When it is implemented, the openInitializationFor: method executes instead of the openInitialization method.

Creating the openInitializationFor: method

To create the openInitializationFor: instance method in the Designer:

1. Select Tools > Methods from the Designer menu to display the Method Editor.
2. Select the Instance radio button.
3. Click New.
4. Type the following method in the Source edit area:
   ```smalltalk
   openInitializationFor: aForm
   "initialize a form that is opening"
   aForm name == mainFormName ifTrue: [
     self mainFormInitialization.
   ].
   aForm == employeeDialog ifTrue: [
     self employeeDialogInitialization.
   ].
   ```
5. Click Save.
6. If the system displays a message box asking whether to subclass the method, click Yes.
Comments about the code. The openInitializationFor: method calls one of the following methods depending on the form that is opening:

- mainFormInitialization
- employeeDialogInitialization

By using a separate method to initialize each form, you make your code easier to read and maintain. You implement code to initialize the main form in “Initializing the main form” on page 374. You implement code to initialize the dialog box in “Initializing the dialog box” on page 394.

Note that the test for each branch is based on a variable rather than on a hard-coded form name. This technique allows you to specify the name in very few places. Later, you can change the name of the form without searching through code for occurrences of the name.

Variable for the main form. The Controller class has an instance variable called mainFormName that stores the name of the controller’s main form.

Variable for the Employee dialog box. The generic way to get an instance of a form is to fetch it from the formDict dictionary in the form’s controller. The following code shows this technique:

```smalltalk
formDict at: #EmployeeDialog.
```

As discussed earlier, it is preferable to refer to a form indirectly, using a variable that is initialized at startup. Once you define the variable, you can test whether the form you are initializing is the same as the variable.

Creating an instance variable

To create the employeeDialog instance variable:

1. Select Tools > Variables from the Designer menu to display the Instance Variables dialog box.
2. Click Add to display the String dialog box.
3. In the Enter instance variable entry field, type:
   ```smalltalk
   employeeDialog
   ```
4. Click OK to close the String dialog box.
5. Click Close to close the Instance Variables dialog box.
Initializing the main form

In this section, you write the mainFormInitialization method, and then you create the four methods that are called by the mainFormInitialization method.

Creating the mainFormInitialization method

Create the mainFormInitialization method with the following code:

```small
mainFormInitialization
    self initVariables.
    self initModel.
    self initMenus.
    self initControls.
```

Again, note how the work of the method is actually carried out by other, smaller methods.

Creating the initVariables method

The initVariables method initializes the controller’s instance variables. Create the method with the following code:

```small
initVariables
    "initialize the controller's instance variables"
    "global variable for inspection"
    InspectorVar := self.
    "variables that store forms"
    employeeDialog := formDict at: #EmployeeDialog.
```

Using the InspectorVar variable

Assigning the InspectorVar variable is a useful debugging technique. Once you have defined it, you can use it as follows:

1. Select File > Test interface from the Designer menu to test the interface.
2. In the Workspace, type and execute the following code:
   ```small
   InspectorVar inspect.
   ```

   The Inspector opens and displays information about the controller.
Creating the initModel method

The application you are building communicates with the Employee table in the ObjectStudio internal database. Create the initModel method, which assigns the model instance variable to the database table:

```smalltalk
initModel
    "initialize the model instance variable"
    | database |
    database := Database getName:
        #EnfinInternalDb.
    model := database getNamed: #Employee.
```

Creating the initMenus method

The initMenus method:

♦ Initializes the Sort and Order menus so that the first item in each menu is checked
♦ Specifies a generic menu handler method, menuAction:, for the Actions menu

Create the initMenus method using the following code:

```smalltalk
initMenus
    "initialize the menus"
    | sortMenu orderMenu |

    sortMenu := cItemDict at: #Sort.
    orderMenu := cItemDict at: #Order.

    "place check marks in first menu item"
    sortMenu setSelectionTo: 1.
    orderMenu setSelectionTo: 1.

    "set up the menu handlers"
    actionDict at: #Actions put: #menuAction:
```
Creating the menuAction: method

The menuAction: method:

♦ Looks up the menu in the cItemDict dictionary.
♦ Gets the value of the selected menu item.
♦ Performs the appropriate action. (Recall that you set up the menu values to enable this programming shortcut.)

Create the menuAction: method with the following code:

```smalltalk
menuAction: aMenuName
   "react to a menu selection in menu aMenuName"
   | menu message |

   "get the menu object"
   menu := cItemDict at: aMenuName.

   "get the menu option"
   message := menu getValue.

   "perform the menu option"
   self perform: message.
```

Creating the initControls method

The initControls method sets the initial state of the controls. For this example, use this method to enable or disable controls when the form opens.

Create the initControls method with the following code:

```smalltalk
initControls
   "initialize the controls"
   self setEnableState.
```
Creating the setEnableState method

The setEnableState method checks whether the Tabular List Box, tableBox, has a selected row. If so, it enables Buttons and menu options. If not, it disables them.

Create the setEnableState method with the following code:

```smalltalk
setEnableState

"Enable or disable buttons and menu options"
| isNotSelected actionMenu |

"initialize temporary variables"
actionMenu := cItemDict at: #Actions.
isNotSelected := tableBox getValue isNil.

"Enable/disable depends only on isNotSelected"
isNotSelected ifTrue: [
  changeButton disable.
  actionMenu disableValue:
    #changeButtonClicked.
  deleteButton disable.
  actionMenu disableValue:
    #deleteButtonClicked.
]
ifFalse: [
  changeButton enable.
  actionMenu enableValue:
    #changeButtonClicked.
  deleteButton enable.
  actionMenu enableValue:
    #deleteButtonClicked.
].
```
Testing your work

Save what you have done so far and test it:

♦ Open the main form and see the list of employees in the table box.
♦ See that the Sort menu has a check mark next to name.
♦ Open the Order cascading menu. There should be a check mark next to Ascending.
♦ Inspect the controller. In Test Interface mode, open the Workspace, type InspectorVar, and select Smalltalk > Inspect.

Sorting the table

This section shows you how to write code that sorts the data that displays in the Tabular List Box. The sorting happens to the table in memory and does not affect the database table.

When you created the Tabular List Box that displays the data, you linked it to the database table, Employee. You may recall from “How MVC works” on page 326 that creating a link in the Designer sets up a dependency relationship that the system uses with the Model View Controller (MVC) mechanism.

Events occur in the following order:

1. The user selects a Sort radio button, changes the Descending check box, or selects an option from the Sort menu.
2. The system responds by resorting the table in memory.
3. The table in memory notifies the table’s dependents that the table changed.
4. The Tabular List Box, a dependent of the table in memory, redispplays itself, reflecting the new sort order.
Using the order: method to sort a table

Tables have a built-in method for sorting themselves. The order: method takes a list of sort options as its parameter. Each sort option is a pair consisting of a field name and an option, either #asc or #desc.

Consider the following examples:

♦ Sort a table (model) ascending by name:
  
  ```small
  model order: (Array with: { #name #asc }).
  ```

♦ Sort a table descending by age:

  ```small
  model order: (Array with: { #age #desc }).
  ```

♦ Sort a table descending by age, then ascending by name:

  ```small
  model order:
    (Array with: { #age #desc }
      with: { #name #asc } ).
  ```

Creating the doSort: method

The doSort: method sorts the table. The parameter is the sort option. Create the method with the following code:

```small
doSort: sortOption
  "sort the table"
  model order: (Array with: sortOption).
```
**Linking the Sort radio buttons**

When the user changes the Sort radio buttons or the Descending check box, the sortButtons method calls the doSort: method.

**Creating the sortButtons method**

The sortButtons method:

1. Gets the value of the sortButtons radio button control. The value is a symbol with the same name as the field to sort, so you do not need to process the value.
2. Gets the value of the Descending check box control, either true or false. You must convert it to #desc or #asc.
3. Creates an array, sortOption, whose elements are the two values in steps 1 and 2.
4. Calls doSort: with sortOption as the argument.

Create the sortButtons method with the following code:

```smalltalk
sortButtons
    "Respond when the user selects a Sort radio button or changes the Descending check box"
    | sortField order sortOption |

sortField := sortButtons getValue.
order := descending getValue
    ifTrue: [ #desc. ]
    ifFalse: [ #asc. ].
sortOption := Array with: sortField with: order.

self doSort: sortOption.
```
Assigning the method to form items

Assign the sortButtons method to the Sort radio buttons and the Descending check box in the Designer. For each form item:

1. Double-click the form item to display the respective Form Radio Buttons Options dialog box or the Form Check Box Options dialog box.
2. Click Methods.
3. Select the sortButtons method from the Available Methods list box.
4. Click Copy.
   
   The method name displays in the Assigned Methods list box.
5. Click OK to close the Method Assignment dialog box.
Linking the sort menus

The sortMenu method calls the doSort: method when the user selects from the Sort or Order menu. The sortMenu method is similar to the sortButtons method, but it gets information from the menus.

Creating the sortMenu method

Create the sortMenu method with the following code:

```smalltalk
sortMenu

"Respond when the user selects from the Sort or Order menus."
| sortField order sortOption |

sortField := (cItemDict at: #Sort) getValue.
order := (cItemDict at: #Order) getValue.
sortOption := Array with: sortField
    with: order.

self doSort: sortOption.
```

Assigning the method to the menus

Assign the sortMenu method to the Sort and Order menus during initialization. Revise the initMenus method to include the code shown in boldface type.

```smalltalk
initMenus

"initialize the menus"
| sortMenu orderMenu |

sortMenu := cItemDict at: #Sort.
orderMenu := cItemDict at: #Order.

"set default check marks"
sortMenu setSelectionTo: 1.
orderMenu setSelectionTo: 1.

"set up the menu handlers"
actionDict at: #Actions put: #menuAction:.
actionDict at: #Sort put: #sortMenu.
actionDict at: #Order put: #sortMenu.
```
Testing your work

Save what you have done so far and test it:

♦ Change a radio button or the Descending check box
♦ Select one of the menu items in the Sort or Order menu

In both cases, the system should re-sort the table according to the criteria you specified.
Synchronizing radio buttons and menus

When you tested the work you did in “Sorting the table” on page 378, you probably noticed that the menus and Radio Buttons are not synchronized. For example, if you select the Age choice from the Sort menu, the system does not update the Age radio button.

To create an elegant user interface, it is important to synchronize controls. Implementing synchronization can be tricky, so you need to examine the options in this section.

Creating methods to synchronize Buttons and menus

At first glance, synchronizing the menus and the sort Buttons seems easy—just reset the menu whenever the Buttons change, and vice versa. This section describes the setSortButtons: method and the setSortMenu: method. The following sections describe how you can get caught in an endless loop and how you can resolve the problem.

Create the following methods:
- The setSortButtons: method resets the Radio Buttons and Check Box after a menu item changes
- The setSortMenu: method resets the Sort and Order menus after a radio button or Check Box changes

Creating the setSortButtons: method

Create the setSortButtons: method with the following code:

```smalltalk
setSortButtons: sortOption
"reset the sort buttons to the sortOption"
| sortField order |

sortField := sortOption at: 1.
order := sortOption at: 2.

sortButtons setSelectionTo: sortField.
descending putValue: (order = #desc).
```
Creating the `setSortMenu:` method

Create the `setSortMenu:` method with the following code:

```plaintext
setSortMenu: sortOption

"reset the sort menu to the sortOption"
| menu sortField order |

sortField := sortOption at: 1.
order := sortOption at: 2.

menu := cItemDict at: #Sort.
menu setSelectionTo: sortField.
menu := cItemDict at: #Order.
menu setSelectionTo: order.
```

Revising the `sortMenu` method

The `sortMenu` method should call the `setSortButtons:` method. Revise `sortMenu` to include the code shown in boldface type.

```plaintext
sortMenu

"Respond when the user selects from the Sort or Order menus."
| sortField order sortOption |

sortField := (cItemDict at: #Sort) getValue.
order := (cItemDict at: #Order) getValue.
sortOption := Array with: sortField
  with: order.

self doSort: sortOption.

"synchronize with the menu by resetting the buttons"
self setSortButtons: sortOption.
```

Revising the `sortButtons` method

Now, revise the `sortButtons` method similarly by calling `setSortMenu:` so that it resets the menus when the Buttons change.
Avoiding the endless loop

If you use the methods described in the previous section, the system enters an endless loop, as shown here:

Setting the value of the Buttons triggers sortButtons, which sets the Sort menu, which triggers sortMenu, which sets the value of the Buttons again.

To avoid the endless loop, you must break the cycle of updates. The next few paragraphs present three possible solutions. The text explains why the first two are not recommended and leads you through the process of implementing the third solution.

Updating the form item

One solution is to bypass the controller item and interact with its corresponding form item. The technique is to send a setSelection message that does not trigger an #Action event.

For example, the original version has the following code:

```
sortButtons setSelectionTo: #name.
```

Using this technique, you would write:

```
sortButtons formItem setSelectionTo: 1.
```

This approach is not recommended because it removes abstraction from the code and the resulting code can be hard to maintain.
Disabling the #Action event

Another solution is to disable the #Action event temporarily while you synchronize. The technique is to manipulate the actionDict for the relevant controls.

The code for this technique is (the code shown in italic type is pseudocode):

```
"save the actions"
savedActions := actionDict at: #sortButtons.

"disable all actions"
actionDict at: #sortButtons put: {}.  

... do the synchronization...

"restore all actions"
actionDict at: #sortButtons put: savedActions.
```

This technique is not recommended, because it produces code that is hard to maintain.

Recommended solution. The recommended solution is to use a flag to track when you are synchronizing. When you start synchronizing, set the flag. Do not update when the flag is set. The pseudocode for this technique is:

```
inSyncMode ifFalse: [

"only do synchronization updates if you are not already synchronizing"

     inSyncMode := true. "set the flag"
     ... do the synchronization...
     inSyncMode := false. "unset the flag"
].
```

This is the recommended approach because it is simple; the code that you write to implement the approach is easy to maintain.
Using flags to synchronize updates

This section describes how to use flags to synchronize updates.

Creating the flag

To create and initialize the flag:
1. Create an instance variable named inSyncMode.
2. Set inSyncMode to false at startup. Revise the initVariables method to include the code shown in boldface type:

```
initVariables
"Initialize the controller’s variables."

"global variable for inspection"
InspectorVar := self.

"variables that store forms"
employeeDialog := formDict at:
   #EmployeeDialog.

"flag to track menu and radio button synchronization"
inSyncMode := false.
```
Modifying the `sortButtons` method to use the flag

The `sortButtons` method should use `inSyncMode`. Revise the `sortButtons` method to include the code shown in boldface type:

```plaintext
sortButtons

"Respond when the user selects a Sort radio button or changes the Descending check box"
| sortField order sortOption |

"only synchronize if you are not already inSyncMode"
inSyncMode ifFalse: [
    inSyncMode := true.
    sortField := sortButtons getValue.
    order := descending getValue
    ifTrue: [ #desc. ]
    ifFalse: [ #asc. ].
    sortOption := Array with: sortField
      with: order.
    self doSort: sortOption.

    "synchronize with the Buttons by resetting the menu"
    self setSortMenu: sortOption.

    inSyncMode := false.
].
```
Modifying the sortMenu method to use the flag

Revise the sortMenu method to include the code shown in boldface type:

```smalltalk
sortMenu
"Respond when the user selects from the Sort or Order menus."
| sortField order sortOption |

"only synchronize if you are not already inSyncMode"
inSyncMode ifFalse: [
    inSyncMode := true.
    sortField := (cItemDict at: #Sort) getValue.
    order := (cItemDict at: #Order) getValue.
    sortOption := Array with: sortField
        with: order.
    self doSort: sortOption.

    "synchronize with the menu by resetting the Buttons"
    self setSortButtons: sortOption.
    inSyncMode := false.
].
```

Testing your work

Save what you have done so far and test it:

♦ Change one of the Radio Buttons or the Descending check box. Then, check that the corresponding menu items change accordingly.

♦ Select a new item in the Sort or Order menu. Check that the corresponding Radio Buttons and the Descending check box change accordingly.
Implementing the Delete option

This section shows you how to implement code to remove a record from the database.

Confirming the deletion

To delete a record from a table, send the removeAt: message to the table, passing the index of the record to delete. For example, the following code deletes the first record:

```smalltalk
model removeAt: 1.
```

Creating the actionDelete method

The actionDelete method checks that a database record is selected and presents a message box to confirm the deletion. The method sends the deleteBoxHandler: message, which performs the actual deletion. Create the method with the following code:

```smalltalk
actionDelete
    "Delete the selected employee record"
    | sel name |

    "proceed only if a record is selected; otherwise exit"
    sel := tableBox getSelection.
    sel = 0 ifTrue: [ ^self. ].

    name := model at: #name.
    MessageBox
        title: 'Table Example'
        text: 'Are you sure you want to delete' 
            ++ name + '?'
        icon: #QuestionMark
        buttons: { #Yes #No}
        action: #deleteBoxHandler: "pass 
            control to the handler"
        controller: self.
```
Deleting the record

If the user confirms the deletion, the deleteBoxHandler: method deletes the selected record.

Creating the deleteBoxHandler: method

Create the deleteBoxHandler: method with the following code:

```smalltalk
deleteBoxHandler: symbol
    "Delete the selected employee record if the user pressed the Yes button."
    symbol = #Yes ifTrue: [
        model removeAt: tableBox getSelection.
    ].
```

Assigning a method to the Delete button

In the Designer, use the Method Assignments dialog box to assign the actionDelete method to the Delete button.

Testing your work

Save what you have done so far and test it. Try selecting a record and clicking Delete.
Displaying the Employee dialog box

This section shows how to open and initialize the Employee dialog box.

Opening the dialog box

To open the Employee dialog box, send the open message to the form; for example:

(formDict at: #EmployeeDialog) open.

As discussed earlier, using the employeeDialog variable you created makes the code more maintainable. A better approach is to use the following code:

employeeDialog open.

Creating the actionAdd and actionChange methods

Create the actionAdd and actionChange methods with the following code:

actionAdd

"open the EmployeeDialog for creating"
employeeDialog open.

ActionChange

"open the EmployeeDialog for updating"

"if there’s no selection, do nothing."
tableBox getSelection = 0 ifTrue: [ ^self. ].
employeeDialog open.

In the Designer, use the Method Assignments dialog box to assign the following on the main form:

♦ actionAdd method to the Add button
♦ actionChange method to the Change button
 Initializing the dialog box

If you test the Add and Change buttons, the system displays the following error:

 Controller does not understand #employeeDialogInitialization

Recall that the openInitializationFor: method sends the employeeDialogInitialization message, which you have not implemented yet. This section shows you how to initialize the dialog box.

 Change mode vs. add mode. There are two modes for displaying the Employee dialog box:

♦ Change mode. You enter change mode when you click Change. In change mode, the system sets up the dialog box with values from the currently selected record.
♦ Add mode. You enter add mode when you click Add. In add mode, the system sets up the dialog box with default values.

Because there are two activities, it is a good idea to create two methods.

 Creating the employeeDialogUpdateFromModel method

The employeeDialogUpdateFromModel method sets up the dialog box for change mode. The method looks up the value of each field in the record that is currently selected and fills in the dialog box with the appropriate information. Create this method with the following code:

 employeeDialogUpdateFromModel

"set up the Employee Dialog box with information from the current employee record."
| isMarried |  
| name put: (model at: #name).
 age put: (model at: #age).
 sex setSelectionTo: (model at: #sex).
 city put: (model at: #city).
 salary put: (model at: #salary).

 isMarried := (model at: #status) == #married
 married putValue: isMarried.

Comments about the code. Initializing the controls, name, age, city, and salary is straightforward—the sex and married controls are trickier. The sex control is a List Box, so you make a selection using setSelectionTo:, rather than setting a value. The married control is a Check Box, so you set its value to true or false, depending on whether the status field is equal to #married.
Creating the employeeDialogClear method

The employeeDialogClear method sets up the dialog box for add mode. The method sets the fields of the dialog box to default values. Create this method with the following code:

```smalltalk
employeeDialogClear
  "set up the Employee dialog using default values"
  | isMarried |

  name put: '???'.
  age put: 21.
  sex setSelectionTo: #female.
  city put: 'Los Angeles'.
  salary put: 30000.

  "the next two lines mirror the code in the
  previous method, but they could be merged into
  one line of code."

  isMarried := false
  married putValue: isMarried.
```

Keeping track of the mode

To keep track of the mode, create an instance variable and track it:
1. Create an instance variable called addMode.
2. Revise the initVariables method to include code shown in boldface type:

```smalltalk
initVariables
  "initialize the controller’s variables"

  "global variable for inspection"
  InspectorVar := self.

  "variables that store forms"
  employeeDialog :=
    formDict at:#EmployeeDialog.

  "flag to track menu and radio button
  synchronization"
  inSyncMode := false.

  "flag that tracks whether you are adding or
  changing a record."

  addMode := false.
```
Creating the employeeDialogInitialization method

The employeeDialogInitialization method tests for add or change mode and sets up the dialog box accordingly. It also sets the label of the Accept button in the dialog box to Add or Change, depending on the mode. Create the method with the following code:

```smalltalk
employeeDialogInitialization
   "initialize the Employee Dialog box, depending on
the value of the addMode flag."

addMode ifTrue: [
   self employeeDialogClear.
   employeeDialogAccept putLabel: 'Add'.
]
ifFalse: [
   self employeeDialogUpdateFromModel.
   employeeDialogAccept putLabel: 'Change'.
].
```

Revising the actionAdd and actionChange methods

Revise the actionAdd and actionChange methods to set the addMode flag. The revisions are shown in boldface type.

```smalltalk
actionAdd
   "open the EmployeeDialog on new employee"
   addMode := true.
   employeeDialog open.

actionChange
   "open the EmployeeDialog on the selected
employee"
   tableView getSelection = 0 ifTrue: [
      ^self.
   ].  "do nothing"
   addMode := false.
   employeeDialog open.
```

Testing your work

Save what you have done so far and test it:

♦ Click Add.
♦ Click Change.

In both cases, the system should display the Employee dialog box, which is initialized appropriately.
Accepting and saving changes

When the user clicks Accept on the Employee dialog box, the system needs to either update or insert a record.

Updating a record

To update a record, use the at:put: message, which sets the value of a field of the current record. For example, the code to set the #name field to John is:

```
model at: #name put: 'John'.
```

Creating the employeeDialogChange method

The employeeDialogChange method gets the value for each field and updates the current record. Create the method with the following code:

```
employeeDialogChange

"accept the changes and update the table"

model at: #name put: (name getValue).
model at: #age put: (age getValue).
model at: #sex put: (sex getValue).
model at: #city put: (city getValue).
model at: #salary put: (salary getValue).
model at: #status put: (married getValue
  ifTrue: [#married.]
  ifFalse: [#single.])
```

Note the use of the single line ifTrue:[...] ifFalse:[...] statement to determine the correct symbol for status.
Inserting a record

In the Table class, a record is implemented as an array of values, and an instance of Table is an array of records.

To add a record to a table, use the add: message. The parameter is the record to add, in the form of an array. The values in the array must display in the exact order as the order in which they are stored.

For example, to add a record to the Employee table, type:

```plaintext
model add: {'John' 32 'male' 'Chicago' 35000 'single'}.
```

If the array argument has the wrong number of elements or has elements in the wrong order, the system issues an error.

Creating the employeeDialogAdd method

The following code is the start of the employeeDialogAdd method. Create the following method:

```plaintext
employeeDialogAdd
   "accept changes; add the record to the table"
   | record |

   record at: 1 put: (name getValue).
   record at: 2 put: (age getValue).
   record at: 3 put: (sex getValue).
   record at: 4 put: (city getValue).
   record at: 5 put: (salary getValue).
   record at: 6 put: (married getValue
      ifTrue: [#married.]
      ifFalse: [#single.]).
   model add: record.

   "set the position pointer of the table"
   model setPositionTo: (model indexOf: record).
```

Comments about the code. The problem with the previous method is that once a record is added, you need to switch to change mode. You can switch to change mode by resetting the addMode flag and then pretending that the dialog box was just opened by calling employeeDialogInitialization again.
Revising the `employeeDialogAdd` method

Revise the `employeeDialogAdd` method to include the code shown in boldface type:

```smalltalk
employeeDialogAdd
   "accept changes; add the record to the table"
   | record |
   record at: 1 put: (name getValue).
   record at: 2 put: (age getValue).
   record at: 3 put: (sex getValue).
   record at: 4 put: (city getValue).
   record at: 5 put: (salary getValue).
   record at: 6 put: (married getValue
      ifTrue: #[married.]
      ifFalse: [#single.]).
   model add: record.
   "set the position pointer of the table"
   model setPositionTo: (model indexOf: record).
   "go to change mode, pretend the dialog was just opened"
   addMode := false.
   self employeeDialogInitialization.
```
Dynamically changing the Accept method

Now, how do you select between the two accept methods? One possibility is to check the addMode flag and choose accordingly.

An alternative, which you will use here, is to dynamically change the method associated with the Accept button. The technique is to change actionDict to point to one of the two methods depending on the addMode.

Revising the employeeDialog Initialization method

Revise the employeeDialogInitialization method to include the code shown in boldface type:

```smalltalk
employeeDialogInitialization

"initialize the Employee Dialog box, depending on
the value of the addMode flag."

addMode ifTrue: [
    self employeeDialogClear.
    employeeDialogAccept putLabel: 'Add'.
    actionDict at: #employeeDialogAccept
        put: #employeeDialogAdd.
].

ifFalse: [
    "update dialog from current record"
    self employeeDialogUpdateFromModel.
    employeeDialogAccept putLabel: 'Change'.
    actionDict at: #employeeDialogAccept
        put: #employeeDialogChange.
].
```

This approach eliminates the need for an intermediate test and branch method.

Testing your work

Save what you have done so far and test it:

♦ Add a record to the table and save it.
♦ Update a record and save it.

In both cases, the Tabular List Box should update itself with the new information.
Implementing the other Buttons

This section explains how to implement the Close, Previous, and Next buttons.

Close button

Closing the dialog box is easy—just send the close message to the dialog box:

1. Create a method with the following code:

   ```
   employeeDialogClose
   "close the Employee dialog box"
   employeeDialogClose.
   ```

2. Use the Method Assignments dialog box to assign the employeeDialogClose method to the Close button.

You may be wondering why you wrote a method to close the dialog box rather than using the Designer to do it. If you use the Designer’s Method Assignments dialog box to assign the close method to a Button and the user clicks that Button, the entire controller closes and the application ends. To close a form other than the main form, you must write a method.
**Previous button**

When the user clicks Previous, the system should decrease the position index of the table and update the dialog box to show the new field values.

To move the table position, send the previous message to the table. The table automatically updates its views, which is, in this case, changing the selection of the Tabular List Box.

aTable previous.

To update the fields from the new current record, you can use the employeeDialogUpdateFromModel method, which you have already written:

1. Create a method with the following code:
   ```ruby
   employeeDialogPrevious
   "move to the previous record"
   model previous.
   self employeeDialogUpdateFromModel.
   
   2. Assign it to the employeeDialogPrevious button.

**Next button**

The process for implementing the Next button is similar to the process for implementing the Previous button:

1. Create a method with the following code:
   ```ruby
   employeeDialogNext
   "move to the next record"
   model next.
   self employeeDialogUpdateFromModel.
   
   2. Assign it to the employeeDialogNext button.
Testing your work

Save what you have done so far and test it:

♦ Save the new record or update the existing record.
♦ Move the current election by clicking Next and Previous.
♦ Close the dialog box by clicking Close.

Optional enhancements

This example still has a few rough edges that are left for you to fix; for example:

♦ When you click Next and Previous, the system is still in add mode. Write code that switches to change mode when you click Next and Previous.

♦ If the user makes changes and either closes the dialog box or clicks Next or Previous, the changes are lost without a warning. Add a message box with the message:
  Unsaved changes: are you sure?

♦ The Accept button (with the label Add or Change) is always enabled. Change the controller so that the Accept button is enabled only if one of the fields is changed.

♦ New records get added to the end of the table. This behavior becomes especially evident when you click Close. Add code to the employeeDialogClose method that sorts the table appropriately and sets the current selection to the record that you just added or changed.
Summary

The example in this chapter demonstrated several user interface development techniques:

♦ Creating menus, including cascading menus
♦ Synchronizing check menus with Radio Buttons
♦ Using openInitialization: when an interface has multiple forms
♦ Opening and closing secondary forms
♦ Creating instance variables
♦ Choosing a method to execute dynamically
♦ Changing a Button label while an application is running

Also, you learned the following techniques for using database tables with a user interface:

♦ Using tables and table links
♦ Adding, changing, and deleting records in tables
♦ Sorting tables

You now should have a good understanding about how to build user interfaces in ObjectStudio.
Index

A
accelerator key events, defining 311
accelerator keys
assigning
methods to 312
to Static Text Fields that are then grouped with Entry Fields 57, 139
creating 312
defining for menus 80
deleting 313
accepts:in: method 285
actionAdd method
creating 393
revising 396
actionChange method
creating 393
revising 396
actionDelete method 391
actionDict dictionary 316
Actions menu 367
activationFor: method 308
add: message 398
addDependent: method 337
addDependent:field: method 337
Alignment toolbar 22
aligning form items 133
displaying 22
resizing form items 133
allDependents method 338
application modal dialog box 59
applications
creating an MDI 272
saving 42
using MVC 326
at:put: message 397
automatic calculation 298
Automation Function Helper 65
B
Bitmap Selector form item 163
bitmaps and forms 53
broadcast method 339
broadcast:with method 339
Button form item
associating with a predefined method 38
creating 36, 37
description 165
linking a method to 41
mnemonics for 139
C
cascading menus, creating 89
Change Form dialog box, exiting 49
changed method 340
changed: method 340
changed:type: method 340
changed:with: method 340
changed:with:with: method 340
Chart form item 167
Check Box form item 172
Check List form item 173
check marks, displaying in menus 82
cItemToLinks dictionary 333
cItemToUpdateLinks dictionary 333
Class Browser
creating controller items 268
creating new interface form items 267
creating Workplace
Template Object 292
classes
creating in the Class Browser 292
DialogBox 301
FormItem 301
MDIWindowsChildForm 272
MessageBox 301
SubForm 272, 301
TabListCtrl 226
close method, closing a dialog box 401
closeRequestedBy: method 308
code generation 12
colors, for forms 53
Combo List form item 174
Component menu 26
conditional icons 228
constants, assigning to variable 298
context-sensitive help, adding to menu items 95
Controller 331
building shell 365
definition 15
initializing 306
MDIWindowsController 272
message protocol 306
overriding default validation 361
Controller item
definition 16
validating data 362
controller items
creating with the Class Browser 268
creating with the Text Editor 268
Controller menu 26
Controller objects in MVC 330
ControllerItem instance 331
to forms 301
naming 370
synchronizing 384
coordinate system of forms 303
copying, form items 127
customizing, form items 137

D
data
  linking to form items 146
validating
  at controller item level 362
  at controller level 361
overview 357
Date Display Format dialog box 140
Date Field form item 175
Debugger 44
Default Button 166
deleteBoxHandler: method 392
dependent objects 326
dependents method 338
Designer
code generation 12
concepts 13
creating
  form items 125
greetMe method 39
  interface component 253
Edit Interface mode. See Edit interface mode
  exiting 24
  interfaces
    saving 72
testing 74
  menu bar 21
  opening 18
  overview 11
  quick tour 31
  starting 32
  status lines, adding 107
Test Interface mode. See Test Interface mode
typical uses 12
using MVC 327
Designer menu  
Component menu 26  
Controller menu 26  
File menu 25  
Form menu 27  
FormItem menu 28  
Help menu 30  
Options menu 30  
Tools menu 29  
dialog box  
application modal 59  
closing 401  
definition 14  
initializing 394  
new 46  
opening 393  
system modal 59  
DialogBox class 301  
dividers. See Moveable Borders  
docking toolbars 100  
doOk method  
assigning 286  
creating 286  
doSort: method 379  
Draw Pad form item 175  
initializing with bitmaps 176  
Drop-down Combo Box form item 185  
Drop-down List Box form item 186  

E  
Edit Interface mode 23, 74  
editDict dictionary 316  
employeeDialogChange method 397  
employeeDialogClear method 395  
employeeDialogInitialization method  
creating 396  
revising 400  
employeeDialogUpdateFromModel method 394  
Entry Field form item  
creating 35  
description 187  

Escape Button 166  
Event Editor  
opening 318  
responding to events 317  
events  
accelerator key 311  
definition 310, 317  
general 317  
on form items 314  
on forms 310  
return key 313  
examples  
creating a Draw Pad 179  
creating a form item  
programmatically 266  
creating Value Sets 242  
database table 363  
font selector interface component 253  
Font Selector interface component 320  
Greet Me 31  
status line 111  
toolbar 104  
workplace application 280  
exting  
Change Form dialog box 49  
Designer 24  
Grid and Snap dialog box 131  
Instance Variables dialog box 271  
Item Groups dialog box 58  
Menu Setup dialog box 78  
Method Assignment dialog box 151  
expression, in Model Editor 295  
external calculation 298
F

File menu 25
flags, synchronizing updates 388
flat toolbars 100
focus, setting to forms 308
font selector
  example
    completing the font selector interface 320
    creating a font selector 253
interface component,
    including in interfaces 263
form events 310
form instances 302
form items
  adding to property pages 115
  aligning 130, 133
  assigning methods 316
  Bitmap Selector 163
  Button 165
  changing
    class 136
    format 139
    options 152
  Chart 167
  Check Box 172
  Check List 173
  Combo List 174
  copying 127
  creating
    mnemonics for 139
    tooltips for 113
    with the Class Browser 267
    with the Designer 34, 125
    with the Text Editor 267
  creating instance variables 128
  customizing 137
  Date Field 175
  definition 16
  deleting 129
  description of class 301
  descriptions 163
  Draw Pad 175
  Drop-down Combo Box 185
  Drop-down List Box 186
  effect of resizing 305
Entry Field 187
events
  assigning methods 150
  defining 314
  geometry 303
  Horizontal Border 190
  key options 155
  Line 190
  linking
    to a method 41
    to data source 146
List Box 190
ListView 193
losing focus 357
miscellaneous options 153
moving 34, 132
Multi Line Entry Field 208
Notebook 208
Number Field 211
OLE Custom Control 215
OLE Item 215
overview 120
Progress Bar 216
Radio Buttons 218
Rectangle 219
resizing 35, 130, 135
Scatter Graph 223
selecting 129
Slider 224
Spin Button 225
Static Text 226
table of 121
Tabular List Box 226
Text Collector 231
Text Editor 233
Topic Box 233
Tree View 234, 235
Value Set 242
vertical border 244
workplace 279
Workplace List Handler 244, 279
Workplace Object Handler 245, 279
Workplace Object Template 245, 279
Form menu 27
Form Toolbar Misc Options
dialog box 103
format, changing a form item 139
FormItem instance 331
FormItem menu 28
forms
adding
   Help menus 83
   menus 75
Window menu 83
assigning to object handler 288
bitmaps 53
changing 49
clearing 47
closing 308
coordinates 303
creating 45
definition 14
deleting 48
description of class 301
gometry 303
naming 50
new 46
opening 307
rectangle 304
resizing, effect of 305
setting
colors 53
focus 47, 308
tab order 55
using grid 130
FormString items, validating
data in 359
Formula text box 295

g
general events 317
generating code 12
geometry of forms 303
getLinkAttributesFor: method 343
getLinkInstancesFor: method 343
getNamed: method 343
getValue method 106
   associating with toolbars 100
object handler 279
template 279
graphs
   linking 300
   setting up in Model Editor 299
greetMe method 41
grid
   lines in a Tabular List Box 227
   using to place form items 130
   grouping items within the tab order 57
H
hasDependent method 338
Help menu 30
help menus, adding to forms 83
help, adding to menu items 95
highlightOff message 166
highlightOn message 166
hitKeyDict dictionary 316
Horizontal Border form item 190
I
in:drop:at: method 286
incremental values. See step value
indexed values box 295
initControls method 376
Initial Focus drop-down list box 50
initializing forms 372
initMenus method 375
initModel method 375
initVariables method 374, 395
Inputs box 295
instance variables
   adding 271
   created by Designer 270
   creating
      for a form item 128
      for Workplace Object Handler 284
   deleting 271
   for dialogs in Controllers 373
Instance Variables dialog box, exiting 271
interface component
adding
to an interface 262
to New Item dialog box 259
to Select Component Type dialog box 260
creating 246, 253
definition 17
effect of changes 262
interfaces
adding interface components 262
creating new 18
definition 15
et mode 74
editing 23
editing 19
initializing 306
making into a template 250
MVC 328
naming 71
saving 72
subclassing 251
testing 23, 43, 74
using a template to create 252
Intermediates box 295
Item Groups dialog box, exiting 58
item traversal in forms 55
items
excluding from the tab order 57
grouping within the tab order 57
including in the tab order 57
rearranging within the tab order 56
ungrouping within the tab order 58

K
keys a form item can respond to 155

L
labels in Chart form items 170
leaf node 234
legend, adding to a Chart 171
Line form item 190
linking objects 332, 334
links
deleting 149
form items to data source 146
read and update 146
updating 149
List Box form item 190
ListView form item 193
lostFocus method 357

M
main form 14, 306
mainFormInitialization method 374
masks 188
MDI
creating an application 272
creating menus 78, 273
forms 14, 52, 272
MDIWindowsChildForm class 272
MDIWindowsController 272
Menu bar, Designer 21
Menu Definition list
demoting items 88
description 86
promoting items 88
Menu editor
changing menu items 88
creating pull-down menus 84
inserting items 87
MDI forms 78, 273
moving items 77
removing items 88
menu items
adding to menus 84
linking to methods 95
Menu Setup dialog box, exiting 78
menuAction: method 376
menus
adding
  context-sensitive help 85, 95
  separators 86
  to forms 75, 79
  to MDI forms 273
allowing multiple selections 82
assigning actions 96
cascading 89
defining
  accelerators 80
  mnemonics 81
displaying check marks 82
not in menu bar 82
pop-up 90
pull-down 84
triggering responses 97
menuSelected: method 97
message box, modal 60
message protocol for controllers 306
MessageBox class 301
messages
  add: 398
  at:put: 397
  highlightOff 166
  highlightOn 166
  previous 402
  setSelection 386
Method Assignment dialog box, exiting 151
Method Editor
  Automation Function Helper 65
  buttons 62
  creating a method reference in 70
  creating and editing methods in 63
  opening from the Designer 39, 61
  pop-up menu 62
  shortcut list boxes and buttons 64
methods
  accepts:in: 285
  actionAdd 393, 396
  actionChange 393, 396
  actionDelete 391
  activationFor: 308
  addDependent: 337
  addDependent:field: 337
  allDependents 338
assigning to a form item 151, 316
associating with a toolbar 106
Automation Function Helper 65
broadcast 339
broadcast:with 339
changed 340
changed: 340
changed:type: 340
changed:with: 340
changed:with:with: 340
close 401
closeRequestedBy: 308
creating
  dependencies 337
  in the Designer 39, 61
deleteBoxHandler: 392
deleting 151, 316
dependents 338
doOk 286
doSort: 379
employeeDialogAdd 398
employeeDialogChange 397
employeeDialogClear 395
employeeDialogInitialization 396, 400
employeeDialogUpdateFrom Model 394
getLinkAttributesFor: 343
getLinkInstancesFor: 343
getNamed: 343
getValue 100, 279
greetMe 41
hasDependent 338
in:drop:at: 286
methods (cont.)
initControls 376
initMenus 375
initModel 375
initVariables 374, 395
linking
to a form item 41
to menu items 95
lostFocus 357
mainFormInitialization 374
menuAction: 376
menus, assigning actions 96
menuSelected: 97
moveNodeFrom:to: 237
NumberCtrl>>characterEvent:
NumberCtrl>>userInputVerification:
open 306, 393
openInitialization 307
openInitializationFor: 307, 372
openMenusForForm:init: 274
order: 379
postOpenInitializationFor:
put: 175, 212
register: 343
release 338
removeDependent:field: 338
removeSelfFromDependents 338
responding to user
interaction 310
return key assigning 287
setEnableState 377
setRangeLower:upper: 217
setSortButtons: 384, 385
setSortMenu: 385
setTextAt:to: 238
sortButtons 380
sortMenu 382
step: 217
stepIt 217
subclassing 285
table of commonly used 309
triggering responses to menu
selections 97
miscellaneous options of form
items 153

mnemonics
creating for form items 139
defining
for Buttons 165
for menus 81
modal dialog box 14, 46, 59, 60
creating 60
types 59
modal message box 60
modality 59
Model Editor
assigning values to variables 298
calculation options 298
creating variables 297
description 295
formula text 295
graphs 299
indexed values 295
inputs 295
intermediates 295
menu bar 297
opening 296
outputs 295
reference 295
undefined variables 295
Model object 332
Model View Controller. See
MVC
models
linking 300
MVC approach to design 324
Moveable Borders
description 156
options 158
moveNodeFrom:to: method
237
Multi Line Entry Field form
item 208
Multiple Document Interface. See
MDI
MVC
   description 324
   interface, building 328
   objects 327
   using in an application 326
   view and controller objects 330

N
New Item dialog box
   adding an interface component 259
node 234
Notebook
   form item 208
   limitations 208
   page, adding items to 210
Number Field form item
   definition 211
   shortcuts 211
   testing user input 213
   validating data 212
NumberCtrl>>characterEvent:
   method 211
NumberCtrl>>userInputVerificationFor:
   method 213
Numeric Display Format dialog box 142

O
Object dependency 326
Object handler, assigning forms 288
objects
   linking between 332
   MVC 327
   notifying of change 339
ObjectStudio Designer. See Designer
OLE Custom Control form item
   Automation Function Helper 215
   description 215
OLE Item form item 215
open method 306
openInitialization method 307
openInitializationFor: method 307, 372
openMenusForForm:init:
   method 274
Options menu 30
options of form items 152
Order cascading menu 368
order: method 379
Outputs box 295

P
passwords, creating Entry Field 187
populating Tree Views 240
pop-up menu 82
   assigning programmatically 294
   attaching to forms 92
   creating 90
   drop-down list box 50
   Method Editor 62
   workplace application 290
postOpenInitializationFor:
   method 307
previous message 402
Progress Bar form item 216
property pages 46
   adding form items to 115
   creating 115
   description 114
   property sheets 46
   accessing from a form 118
   creating 116
   description 114
   editing 117
   protocol for controller messages 306
   pull-down menus, creating 84
   put: method 175, 212

R
Radio Button form item 218
read links 146
Rectangle form item 219
rectangle of forms 304
Reference box 295
register: method 343
regular link 332
release method 338
removeDependent:field:
   method 338
removeSelfFromDependents
   method 338
resize items on forms. See
Moveable Borders
resizing, form items 135
return key events 313
return key method
assigning 287
defining 313
deleting 314
rows
deleting from tables 391
inserting in tables 398
updating 397

S
Scatter Graph form item 223
Select Component Type dialog box, adding an interface component 260
Select Controller Type dialog box, adding an interface 250
selection heads, definition 129
separators, adding to menus 86
setEnableState method 377
setRangeLower:upper: method 217
setSelection message 386
setSortButtons: method 384, 385
setSortMenu: method 385
setTextAt:to: method 238
shortcuts, Number Field 211
Slider form item 224
Smalltalk
Designer 15
form items 16
forms 14
Sort menu 368
sortButtons method 380
sortMenu method 382
source data, position in Chart form item 169
Spin Button form item 225
split list in a Tabular List Box 226
start value, in model graphs 299
starting the Designer 32

Static Text form item 226
creating in Greet Me example 34
creating mnemonics for 139
status line 22
creating 107
displaying 22
editing 109
example 111
step value, in model graphs 299
step: method 217
stepIt method 217
stop value, in model graphs 299
String Display Format dialog box 144
subclassing
before a form opens 307
closing a form 308
interface 251
method 285
when a form receives focus 308
SubForm class 272, 301
subforms
description 14
new 46
synchronized List Boxes 191
Synchronized List Groups dialog box 192
synchronized Radio Buttons and menus 384
system modal dialog box 14, 46, 59
tab order in forms 55

tables
deleting rows 391
inserting rows 398
sorting 378
updating 397
TabListCtrl class 226
Tabular List Box form item 226

template
changeable 248
creating 250
creating object for workplace applications 292
fixed 248
types for interfaces 247
using to create new interfaces 252
Test Interface mode 23, 43, 74
Text Collector form item 231
Text Editor
creating
controller items 268
new interface form items 267
form item 233
pop-up menu 62
Toolbar Definition list box
adding items 102
description 101
Toolbar Options dialog box 99
toolbars
alignment 133
associating a method with 106
check style 100
creating 98
displaying 22
docking 100
editing 99
example 104
flat 100
items 100
miscellaneous options 103
tooltips 100
value 100
Tools menu 29

tooltips
for Buttons on toolbars 100
for form items 113

Topic Box form item
description 233
mnemonics for 139
Tree View form item 234, 235
Tree Views, populating 240

undefined variables box 295
ungrouping items within the tab order 58
update links 146, 332, 336
user interaction, responding to 310
userInputVerificationFor:
method in class NumberCtrl 213
userInputVerificationFor:value : method 213

validating data
at controller item level 362
at controller level 361
Date Fields 175
Number Fields 212
overview 357
Value Set form item, creating 242
values
assigning to variables 298
incremented in model graph 299
variables
assigning values 298
creating 297
input 295
input on graphs 299
instance, creating 284
intermediate 295
output 295
output on graphs 299
undefined 295
vertical border form item 244
view objects
in MVC 330
link between model object 332
views, separating from model 324
Window menu
  adding to an MDI form 275
  adding to forms 83
  adding to other forms 276
workplace applications
  actions 278
  adding form items 281
  changing form items 282
creating
  editor form 283
  main form 281
  template object 292
description 277
dynamic option 279
example 280
pop-up menus 290
workplace form items 279
Workplace List Handler form item 244, 279
Workplace Object Handler form item 245, 279
Workplace Object Template form item 245, 279
Workplace Tabular List Boxes 228
Workplace Template Object creating
  a new class 292
  a workplace template 293