Using Zen Components

Version 2018.1
2024-05-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Zen Components
Caché Version 2018.1 2024-05-02
Copyright © 2024 InterSystems Corporation
All rights reserved.

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble® InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™ HealthShare® Health Connect Cloud™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book **1**
- Zen Attribute Data Types 1
- Zen Component Event Handlers 2

1 Zen Tables **3**
- 1.1 <tablePane> 4
- 1.2 Data Sources 5
  - 1.2.1 Specifying an SQL Query 5
  - 1.2.2 Generating an SQL Query 6
  - 1.2.3 Referencing a Class Query 8
  - 1.2.4 Using a Callback Method 8
  - 1.2.5 Changing the Data Source Programmatically 11
- 1.3 Query Parameters 12
- 1.4 Table Columns 13
  - 1.4.1 colName 15
  - 1.4.2 OnDrawCell 15
- 1.5 Table Style 16
- 1.6 Conditional Style for Rows or Cells 17
- 1.7 Snapshot Mode 19
  - 1.7.1 Fetching Data From the Server 20
  - 1.7.2 Navigating Snapshot Tables 20
- 1.8 Column Filters 21
- 1.9 Column Links 25
- 1.10 User Interactions 26
  - 1.10.1 Navigation Buttons 26
  - 1.10.2 Navigation Keys 27
  - 1.10.3 Sorting Tables 27
  - 1.10.4 Selecting Rows and Columns 28
- 1.11 Table Refresh 29
- 1.12 Table Touchups 30
  - 1.12.1 Data Values 30
  - 1.12.2 Header and Body Alignment 30

2 Zen and SVG **31**
- 2.1 Fonts for SVG 31
- 2.2 SVG Component Layout 32
  - 2.2.1 <svgFrame> 32
  - 2.2.2 <svgGroup> 37
  - 2.2.3 <svgSpacer> 37
  - 2.2.4 <rect> 37
- 2.3 SVG Component Attributes 38
- 2.4 Meters 39
  - 2.4.1 Providing Data for Meters 39
  - 2.4.2 Meter Attributes 40
  - 2.4.3 <fuelGauge> 41
  - 2.4.4 <indicatorLamp> 42
  - 2.4.5 <lightBar> 43
  - 2.4.6 <slider> 44

2.4.7 <smiley> 45
2.4.8 <speedometer> 45
2.4.9 <trafficLight> 46
2.5 Charts 47
2.6 <radialNavigator> 47
2.7 <ownerDraw> 49

3 Zen Charts 51
3.1 Types of Chart 51
3.1.1 Bar Charts 52
3.1.2 Bubble Charts 53
3.1.3 Bullseye Charts 56
3.1.4 Combo Charts 57
3.1.5 Difference Charts 59
3.1.6 High/Low Charts 60
3.1.7 Line Charts 61
3.1.8 Percent Bar Charts 62
3.1.9 Pie Charts 63
3.1.10 Scatter Diagrams 66
3.1.11 Tree Map Charts 68
3.2 Providing Data for Zen Page Charts 68
3.2.1 Using a JavaScript Method 69
3.2.2 Using a Data Controller 73
3.2.3 Limiting the Data Set 73
3.3 Chart Layout, Style, and Behavior 74
3.3.1 Specifying Size and Position 75
3.3.2 Layout and Style 76
3.3.3 Plot Area 76
3.3.4 Markers 82
3.3.5 Legends 83
3.3.6 Titles 84
3.3.7 User Selections 85
3.4 Chart Axes 85

4 Zen Forms 89
4.1 Forms and Controls 90
4.2 User Interactions 91
4.3 Defining a Form 92
4.4 Providing Values for a Form 95
4.5 Detecting Modifications to the Form 95
4.6 Validating a Form 96
4.7 Errors and Invalid Values 97
4.8 Processing a Form Submit 97
4.9 User Login Forms 98
4.10 Dynamic Forms 99

5 Zen Controls 101
5.1 Control Attributes 101
5.2 Data Drag and Drop 104
5.3 Control Methods 105
5.4 Buttons 105
5.4.1 <button> 105

5.4.2 <image> ... 106
5.4.3 <submit> ... 108
5.5 Text ... 108
5.5.1 <label> ... 108
5.5.2 <text> ... 108
5.5.3 <textarea> ... 109
5.5.4 <password> ... 110
5.6 Selections ... 110
5.6.1 <checkbox> ... 110
5.6.2 <multiSelectSet> ... 111
5.6.3 <fileUpload> ... 113
5.6.4 <colorPicker> ... 114
5.6.5 <radioSet> ... 115
5.6.6 <radioButton> ... 117
5.7 Lists ... 117
5.7.1 <select> ... 118
5.7.2 <listBox> ... 121
5.7.3 <dataListBox> ... 122
5.7.4 <combobox> ... 125
5.7.5 <dataCombo> ... 127
5.7.6 <lookup> ... 134
5.8 Dates ... 137
5.8.1 <calendar> ... 137
5.8.2 <dateSelect> ... 139
5.8.3 <dateText> ... 141
5.9 Grid ... 144
5.9.1 <dynaGrid> ... 144
5.9.2 <dataGrid> ... 153
5.10 Hidden ... 165

6 Model View Controller ... 167
6.1 Model ... 168
6.1.1 %ZEN.DataModel.ObjectDataModel ... 169
6.1.2 %ZEN.DataModel.Adaptor ... 170
6.2 Controller ... 170
6.2.1 <dataController> ... 171
6.2.2 <dataController> Attributes ... 171
6.2.3 <dataController> Methods ... 173
6.3 View ... 174
6.3.1 Data View Attributes ... 174
6.3.2 The Controller Object ... 175
6.3.3 Multiple Data Views ... 175
6.4 Constructing a Model ... 176
6.4.1 Step 1: Type of Model ... 176
6.4.2 Step 2: Object Data Model ... 176
6.5 Binding a <form> to an Object Data Model ... 178
6.5.1 Step 1: Data Controller ... 178
6.5.2 Step 2: Data View ... 179
6.5.3 Step 3: Initial Results ... 179
6.5.4 Step 4: Saving the Form ... 180
6.5.5 Step 5: Performing Client-side Validation ... 181

6.5.6 Step 6: Setting Values Programmatically ... 181
6.6 Adding Behavior to the <form> ... 181
6.6.1 Step 1: Opening a New Record ... 181
6.6.2 Step 2: Creating and Deleting Records ... 183
6.7 <dynaForm> with an Object Data Model ... 185
6.7.1 Step 1: <dynaForm> is Easy ... 185
6.7.2 Step 2: Converting to <dynaForm> ... 186
6.7.3 Step 3: Automatic Control Selection ... 188
6.8 <dynaForm> with an Adaptor Data Model ... 191
6.8.1 Step 1: Generating the Form ... 191
6.8.2 Step 2: Property Parameters ... 192
6.8.3 Step 3: Adding Behavior to the <dynaForm> ... 193
6.8.4 Step 4: Virtual Properties ... 195
6.9 Data Model Classes ... 197
6.9.1 Data Model Class Properties ... 198
6.9.2 Data Model Class Parameters ... 198
6.9.3 Data Model Property Parameters ... 198
6.9.4 Value Lists and Display Lists ... 200
6.9.5 Object Data Model Callback Methods ... 201
6.9.6 Virtual Properties ... 201
6.9.7 Controller Actions ... 203
6.9.8 Data Model Series ... 203
6.9.9 Custom Data Model Classes ... 204

7 Navigation Components ... 207
7.1 Links ... 209
7.1.1 <link> ... 209
7.1.2 <locatorBar> ... 210
7.1.3 <locatorLink> ... 213
7.2 Menus ... 213
7.2.1 <menuItem> ... 215
7.2.2 <menu>, <hmenu>, and <vmenu> ... 216
7.2.3 <menuSeparator> ... 216
7.2.4 <accordionMenu> ... 217
7.3 Navigator ... 218
7.3.1 Creating and Sizing a <navigator> ... 220
7.3.2 Adding Content to the Navigator ... 220
7.3.3 Changing the Display and Appearance of Items ... 223
7.3.4 Editing Values in Items ... 223
7.3.5 Creating a Multiple Choice Item ... 224
7.3.6 Displaying HTML ... 225
7.4 Toolbar ... 225
7.5 Tabs ... 227
7.5.1 <tabGroup> ... 227
7.5.2 <lookoutMenu> ... 229
7.5.3 <tab> ... 231
7.6 Trees ... 232
7.6.1 <expando> ... 232
7.6.2 <dynaTree> ... 236
7.7 Filters ... 243
7.7.1 <buttonView> ... 245

8 Popup Windows and Dialogs 249
8.1 Modal Groups 249
8.1.1 Static Modal Groups 250
8.1.2 Dynamic Modal Groups 252
8.1.3 Built-in Modal Groups 254
8.1.4 The show Method 257
8.1.5 <modalGroup> Attributes 258
8.2 Popup Windows 259
8.3 Dialogs 262
8.3.1 File Selection Dialog Window 262
8.3.2 Color Selection Dialog Window 263
8.3.3 Search Dialog Window 263
8.3.4 Creating a Dialog Window 264
8.3.5 Creating a Dialog Window Template 265

9 Other Zen Components 267
9.1 HTML Content 267
9.2 Framed Content 269
9.2.1 <iframe> Attributes 269
9.2.2 Images as Button Controls 269
9.2.3 Rendering Image Data Streams 270
9.3 Timer 271
9.4 Field Sets 271
9.5 Color Selector 273
9.6 Color Wheel 274
9.7 Repeating Group 275
9.8 Dynamic View 277
9.8.1 <dynaView> OnGetViewContents Callback Method 278
9.8.2 <dynaView> Attributes 279
9.9 Schedule Calendar 280
9.9.1 <schedulePane> OnGetScheduleInfo Callback Method 280
9.9.2 <schedulePane> Attributes 282
9.10 Finder Pane 286

List of Figures

Figure 3–1: Bar Chart 52
Figure 3–2: Radius Data Series 53
Figure 3–3: Color Data Series 54
Figure 3–4: Opacity Data Series 55
Figure 3–5: Bullseye Chart 56
Figure 3–6: Combo Chart Displaying Area, Bar and Line Charts 57
Figure 3–7: Combo Chart Displaying Four Area Charts 57
Figure 3–8: Combo Chart with Target Lines 58
Figure 3–9: Difference Chart 59
Figure 3–10: Area Chart 59
Figure 3–11: High/Low Chart 60
Figure 3–12: Line Chart 61
Figure 3–13: Percent Bar Chart 63
Figure 3–14: Pie Chart with One Data Series 63
Figure 3–15: How Zen Plots Pie Charts by Item 65
Figure 3–16: How Zen Plots Pie Charts by Series 65
Figure 3–17: Zen Pie Chart from Both Items and Series 66
Figure 3–18: XY or Scatter Chart 66
Figure 3–19: Tree Map Chart 68
Figure 3–20: Data Series Count and Size 74
Figure 3–21: Layout Attributes for Zen Charts 75
Figure 3–22: Pivoted Bar Chart 80
Figure 3–23: Stacked Bar Chart 81
Figure 3–24: Stacked Line Chart 81
Figure 3–25: Line Chart Displayed as Multiples 81
Figure 3–26: Time Based Line Chart 82
Figure 3–27: Line Chart with Two y Axes 86
Figure 3–28: Same data Plotted on One y-axis 86
Figure 4–1: Class Inheritance Among Form and Control Components 91
Figure 5–1: Data Model for the Dynamic Grid Control 146
Figure 5–2: Data Grid Layout 157
Figure 5–3: Ascending and Descending Sort Order 161
Figure 6–1: Model View Controller Architecture 168
Figure 6–2: Data Model Classes 169
Figure 6–3: Data Controller and Data View Classes 171
Figure 6–4: Data Model with Name-Value Pairs 203
Figure 6–5: Data Model with Data Series 204
Figure 7–1: Zen Navigation Components 208

List of Tables

Table 1–1: XML Entities for Use in sql Attribute Values 6
Table 1–2: QueryInfo Properties 11
Table 1–3: <condition> predicate Values 18
Table 2–1: SVG Component Attributes 38
Table 2–2: Meter Component Attributes 40
Table 3–1: Chart Layout and Style Attributes 76
Table 3–2: Chart Axis Attributes 87
Table 4–1: Form Component Attributes 92
Table 4–2: Form Submit Sequence 98
Table 5–1: Control Component Attributes 102
Table 5–2: List Box Component Attributes 122
Table 5–3: Combo Box Component Attributes 126
Table 5–4: XML Entities for Use in sqlLookup Attribute Values 130
Table 5–5: <dataCombo> Display Sequence 133
Table 6–1: <dynaForm> Controls Based on Data Types 190
Table 6–2: Data Model Class Parameters 198
Table 6–3: Data Model Property Parameters 199
Table 6–4: Object Data Model Callback Methods 201
Table 6–5: Custom Data Model Class Methods 205
Table 7–1: Menu Cell Attributes 215
Table 7–2: Tab Group Attributes 229
Table 8–1: Client Side Methods for Controlling Popup Windows 259

About This Book

This book describes how to use the built-in components that Zen provides for laying out tables, charts, forms, menus, dialogs, and other items for web applications.

This book contains the following sections:

- "Zen Tables" explains how to display the results of a database query as an HTML table.
- "Zen and SVG" describes how to use Scalable Vector Graphics (SVG) to display data-driven charts and meters.
- "Zen Charts" explains how to place a chart on a Zen page.
- "Zen Forms" explains how to lay out a form that allows a user to edit data.
- "Zen Controls" describes the user controls that you can place on a form.
- "Model View Controller" explains how the MVC model can assist the flow of data to a Zen page.
- "Navigation Components" describes menu and link components that support user navigation.
- "Popup Windows and Dialogs" describes components that display their contents over the main application page.
- "Other Zen Components" describes built-in components that do not fit into the categories listed above.

There is also a detailed table of contents.

The following books provide related information:

- *Using Zen* provides the conceptual foundation for developing web applications using Zen.
- *Using JSON in Caché* describes how to use the %Object and %Array classes, which provide support for JSON, which is a useful format to ship data between client and server.
- *Developing Zen Applications* explores programming issues and explains how to extend the Zen component library with custom code and client-side components.
- *Using Zen Reports* explains how to generate reports in XHTML and PDF formats based on data stored in Caché.

For general information, see *Using InterSystems Documentation*.

Zen Attribute Data Types

Many attributes of Zen objects have one of the following underlying data types:

- %ZEN.Datatype.boolean which has the value "true" or "false" in XData Contents, 1 or 0 in server-side code, true or false in client-side code.
- %ZEN.Datatype.caption. which makes it easy to localize text into other languages, as long as a language DOMAIN parameter is defined in the Zen page class. The %ZEN.Datatype.caption data type also enables you to use $$$Text macros when you assign values to the property from client-side or server-side code.

Zen Component Event Handlers

Many Zen components have attributes with names that begin with “on...”. These attributes identify event handlers for the component. The value of the attribute is a client-side JavaScript expression that Zen invokes when the related event occurs. This expression generally invokes a client-side JavaScript method defined in the page class. This method is the “handler” for the event.

When providing a value for an event-handler attribute, use double quotes to enclose the value and single quotes (if needed) within the JavaScript expression. For example:

```
<svgFrame ondragCanvas="alert('HEY');"/>
```

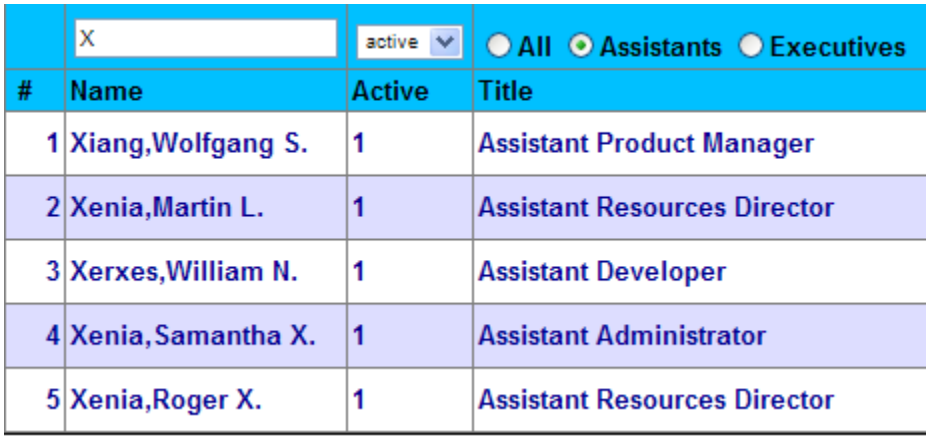
The JavaScript expression can contain Zen #()# runtime expressions.

1

Zen Tables

A Zen table is data-driven. It takes the resultset returned by an SQL query and displays it as an HTML table. There are a number of ways to generate the resultset for a Zen table. You can specify an SQL statement, reference a predefined SQL query, or provide Zen with the inputs it needs to generate an SQL query for you.

Once you have the data, you can style the resulting table in any way you wish. The following figure shows a simple example. This table uses “zebra” patterning for alternate rows. The user has entered data in the table header to filter the results that the table displays. In this case, the user has selected only entries whose names begin with X, with Active status, who are also Assistants.



#	Name	Active	Title
1	Xiang,Wolfgang S.	1	Assistant Product Manager
2	Xenia,Martin L.	1	Assistant Resources Director
3	Xerxes,William N.	1	Assistant Developer
4	Xenia,Samantha X.	1	Assistant Administrator
5	Xenia,Roger X.	1	Assistant Resources Director

This chapter explains how to work with Zen tables as follows:

- How to place a table on a Zen page
- How to identify the data source for a Zen table
- How to supply parameters for the table query, if needed
- How to specify column details, including filters and links
- General style properties that Zen tables can have
- How to define data-specific styling for rows and columns
- How snapshot mode supports multipage tables and simplifies refresh operations
- How a Zen table handles user interactions
- What happens during table refresh operations, and how to request them

1.1 <tablePane>

<tablePane> is the XML projection of the versatile %ZEN.Component.tablePane class. To place a table on a Zen page, place a <tablePane> component inside the page class XData Contents block.

This chapter describes the various component and auxiliary classes that Zen supplies to support tables. The following list summarizes the XML elements used to represent these classes in XData Contents. The most important of these is <tablePane>:

- "<tablePane>" — Draws an HTML table based on an SQL query. Each row in the resultset is displayed as a row in the table. A <tablePane> may contain the following elements, as needed:
  - "<parameter>" — Each <parameter> element provides one of the parameters required to construct the <tablePane> query.
  - "<column>" — Each <column> element specifies layout, style, and behavior details for a particular column in the resulting table. <column> elements are optional when the table displays all columns in the resultset. <column> elements are required when a <tablePane> needs to select which columns in the resultset to display.
  - "<condition>" — Each <condition> element defines one data-specific detail that applies to rows and cells within the table. For example, cells that contain a certain value might display a certain background color, such as red to indicate an error condition. The specific cells that contain this value might be different each time the table refreshes. Zen keeps track of these details for you and colors all cells appropriately.
- "<tableNavigator>" — Automatically provides a standard set of buttons for moving through the pages of a multipage table.
- "<tableNavigatorBar>" — An alternative to <tableNavigator>, this element provides extra buttons to help users navigate large, multipage tables.

<tablePane> has the following general-purpose attributes.

Attribute	Description
dataSource	Specifies which columns from the %ResultSet to display, and in what order. Possible values are: • "query" — All columns referenced by the query appear, in order from left to right. • "columns" — Only the columns explicitly defined as <column> entries within the <tablePane> appear, in order from left to right. When you omit *dataSource* from the <tablePane>, Zen uses the value "query" by default, unless there are <column> entries defined, in which case Zen ignores any *dataSource* value and uses "columns".
initialExecute	If *initialExecute* is true, the <tablePane> executes the associated query when the table first displays. Otherwise the <tablePane> executes the query only on demand. The default is true. *initialExecute* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

Attribute	Description
unlockSession	Indicates whether updates to the <tablePane> component should unlock the CSP session while retrieving data from the server. Setting this attribute to true makes it easier to maintain a responsive UI while running slower queries on the server. In general, you should use this feature when the server does not update the session data. The default value is false. *unlockSession* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

<tablePane> offers many additional attributes that you can use to configure layout, style, and behavior. The following topics describe them.

1.2 Data Sources

A <tablePane> element must indicate a data source for the table in one of the following ways:

- "Specifying an SQL Query"
- "Generating an SQL Query" by providing a simple description that Zen uses to generate an SQL statement
- "Referencing a Class Query" to obtain a result set
- "Using a Callback Method" to obtain a result set
- "Changing the Data Source Programmatically" for a table at runtime

The next several topics describe each option in detail.

Regardless of which option you use, all techniques support the *maxRows* attribute. It controls the size of the data returned. The following table provides *maxRows* details.

Attribute	Description
maxRows	The maximum number of rows to fetch. For ordinary tables this is the maximum number of rows to display. For snapshot tables, *maxRows* is the maximum size of the snapshot and *pageSize* is the number of rows to display per page. The default value for *maxRows* is 100. The <radioSet>, <select>, <dataListBox>, <dataCombo>, and <repeatingGroup> components also support the *maxRows* attribute.

1.2.1 Specifying an SQL Query

A <tablePane> can provide a complete SQL statement as the value of its *sql* attribute. For example:

XML

```
<tablePane id="table"
           sql="SELECT ID,Name FROM MyApp.Employees
                WHERE Name %STARTSWITH ? ORDER BY Name"
           >
     <parameter value="Z"/>
</tablePane>
```

The following table provides *sql* details.

Attribute	Description
sql	The value of this attribute is a complete SQL statement, which Zen executes at runtime to provide the contents of the table. The <radioSet>, <select>, <dataListBox>, <dataCombo>, and <repeatingGroup> components also support the *sql* attribute. You may provide any input parameter values for the SQL query by placing <parameter> elements inside the <tablePane> container. For details, see the section "Query Parameters."

The *sql* attribute is the XML projection of the %ZEN.Component.tablePane property sql. Therefore, the *sql* attribute value must escape any XML special characters. For example, in place of the less-than symbol < you must substitute the XML entity `<` as follows:

```
sql="select * from infonet_daten.abopos where lieferadresse=? and status<9"
```

The following table lists XML special characters that cause problems when they appear in *sql* strings, and the XML entities to substitute for them.

Table 1–1: XML Entities for Use in sql Attribute Values

Character	XML Entity	Description
>	>	Right angle bracket or "greater than" symbol.
<	<	Left angle bracket or "less than" symbol.
&	&	Ampersand.
'	'	Single quotation mark or apostrophe. A string enclosed in single quotes needs the ' entity to represent the ' character.
"	"	Double quotation mark. A string enclosed in double quotes needs the " entity to represent the " character.

Unlike most other %ZEN.Component.tablePane properties, you cannot set the sql property from the client at runtime. You can set it only from XData Contents. This is because *sql* is an encrypted attribute. The *sql* attribute value is encrypted (using the current session key) when it is sent to the client. If this value is returned to the server, it is automatically decrypted.

This prevents users from seeing the definition of an SQL statement if they view page source within their browser and prevents client logic from constructing arbitrary queries. For security reasons, query activities should always be restricted to the server.

1.2.2 Generating an SQL Query

The <tablePane> component supports attributes that allow you to automatically generate the query based on a simple description. This approach is similar to using a callback method, as described in a later topic, except that Zen generates the callback method for you, based on your description of the query. <dataListBox> and <dataCombo> also supports these attributes.

Attribute	Description
groupByClause	An SQL GROUP BY clause such as `"Year,State"`. The *groupByClause* value can be a literal string, or it can contain a Zen #()# runtime expression.

Attribute	Description
orderByClause	An SQL ORDER BY clause such as `"Name,State"`. If not provided, then whenever the user clicks on a column header, the next query contains the appropriate ORDER BY clause based on the user's choice. The *orderByClause* value can be a literal string, or it can contain a Zen #()# runtime expression.
tableName	The name of the SQL table that provides the data for the table. This value is used in the FROM clause for the generated query. The *tableName* value can be a literal string, or it can contain a Zen #()# runtime expression.
whereClause	An SQL WHERE clause such as `"Name='Elvis'"`. When a *whereClause* is provided in the <tablePane> definition in XData Contents, this sets an initial value for the whereClause property of the tablePane object. If client-side or server-side code later changes the value of this whereClause property, the new value overrides the original value. This means you can initially set up a table to show only certain values, but another line of code can change the whereClause value, causing your users to see a different set of values when it refreshes. For example, if any column filters are defined in this table, Zen dynamically creates a WHERE clause for the <tablePane> based on the current filter values selected by the user. The *whereClause* value can be a literal string, or it can contain a Zen #()# runtime expression.

To have the <tablePane> generate an SQL query, you must do the following:

1. Provide a value for the *tableName* attribute.
2. Do not provide values for the *sql*, *queryClass*, *queryName*, or *OnCreateResultSet*. All of these attributes take precedence over the behavior described in this topic.

   After you satisfy the first two conditions, Zen assumes a "columns" value for *dataSource*.
3. Define the names of one or more columns by providing <column> elements inside the <tablePane>. You must define at least one column or Zen generates a "Missing SELECT list" error.
4. You can add query parameters by providing <parameter> elements inside the <tablePane>.
5. You can add clauses for the generated query by providing the <tablePane> attributes *groupByClause*, *orderByClause*, or *whereClause*, or by allowing defaults to prevail as described in the table above.

The following is a simple example:

XML

```
<tablePane id="table"
    tableName="MyApp.Employee">
    <column colName="ID" hidden="true"/>
    <column colName="Name"/>
</tablePane>
```

This <tablePane> example generates an SQL statement similar to the following:

```
SELECT ID,Name FROM MyApp.Employee
```

Zen executes this query to provide the contents of the table. In the example, the ID column is marked as *hidden*. This means that its value is fetched (it can be used for conditions or actions) but not displayed. For details about *hidden* and other <column> attributes, see the section "Table Columns."

Note: A generated SQL query can be useful for tables with column filters.

1.2.3 Referencing a Class Query

A <tablePane> can reference a pre-existing class query to obtain a %ResultSet object. The following components also support this approach: <radioSet>, <select>, <dataListBox>, <dataCombo>, and <repeatingGroup>.

Attribute	Description
queryClass	The name of the class containing the query. You must also provide a value for *queryName*.
queryName	The name of the class query that provides the %ResultSet for this <tablePane>. You must also provide a value for *queryClass*.

You may provide any input parameter values for the query by placing <parameter> elements inside the <tablePane>. For example:

XML

```
<tablePane id="table"
           queryClass="MyApp.Employee"
           queryName="ListEmployees">
    <parameter value="Sales"/>
    <parameter value="NEW YORK"/>
</tablePane>
```

The value of the parameter in the <tablePane> is the value used to create the %ResultSet object. It overrides any default value set in the class query in all cases.

1.2.4 Using a Callback Method

A <tablePane> can use a callback method to obtain a %ResultSet object. The following <tablePane> attributes support this approach. <dataListBox>, <dataCombo>, and <altJSONSQLProvider> also support these attributes.

Attribute	Description
OnCreateResultSet	Name of a server-side callback method in the Zen page class. For more information, see OnCreateResultSet.
OnExecuteResultSet	Name of a server-side callback method in the Zen page class. For more information, see OnExecuteResultSet.

Attribute	Description
showQuery	*showQuery* works only if an *OnCreateResultSet* callback is used to generate the table, and only if this callback sets the queryText property of the QueryInfo object to contain the text of the query. Of the various components that use callback methods to generate SQL queries, only <tablePane> and <dataCombo> support the *showQuery* attribute. If *showQuery* is true, the Zen page displays the SQL query used to provide the contents of the <tablePane> or <dataCombo> component. This is useful for troubleshooting purposes, during application development. The default *showQuery* value is false. *showQuery* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types." The *showQuery* value can be a literal string, or it can contain a Zen #()# runtime expression.

1.2.4.1 OnCreateResultSet

Name of a server-side callback method in the Zen page class. This method instantiates the %ResultSet object and set any %ZEN.Auxiliary.QueryInfo "QueryInfo Properties" appropriately. Zen invokes this method whenever it draws the table, automatically passing it the following parameters:

- %Status — an output parameter for method status
- %ZEN.Auxiliary.QueryInfo — the QueryInfo object for the <tablePane>

The callback must return a %ResultSet object. The following is a valid method signature. Also see the detailed example following this table:

Class Member

```
Method CreateRS(Output tSC As %Status,
     pInfo As %ZEN.Auxiliary.QueryInfo)
     As %ResultSet
{ }
```

To use the above method as the callback, the developer would set `OnCreateResultSet="CreateRS"` for the <tablePane>.

If defined, *OnCreateResultSet* takes precedence over any other techniques for providing data for a <tablePane>.

1.2.4.2 OnExecuteResultSet

Name of a server-side callback method in the Zen page class. This method executes the %ResultSet object returned by the *OnCreateResultSet* callback. Zen automatically invokes the *OnExecuteResultSet* callback after the *OnCreateResultSet* callback, passing it the following parameters:

- %ResultSet — the result set from *OnCreateResultSet*
- %Status — an output parameter for method status
- %ZEN.Auxiliary.QueryInfo — the QueryInfo object for the <tablePane>

Optionally, you can suppress invocation of the *OnExecuteResultSet* callback by setting the QueryInfo queryExecuted property to true at the end of the *OnCreateResultSet* callback.

The *OnExecuteResultSet* callback must return a%ZEN.Datatype.boolean indicating whether or not the result set was executed. The following is a valid signature for this callback:

Class Member

```
Method ExecuteRS(myRS As %ResultSet,
     Output pSC As %Status,
     pInfo As %ZEN.Auxiliary.QueryInfo)
     As %Boolean
{ }
```

To use the above method as the callback, the developer would set `OnExecuteResultSet="ExecuteRS"` for the <tablePane>.

1.2.4.3 Callback Example

The following is a detailed example of using a callback method to create a %ResultSet for a tablePane. The callback method constructs a dynamic SQL statement in response to current values of tablePane properties sortOrder and sortColumn. The tablePane automatically passes these values to the callback method as the corresponding properties of the input %ZEN.Auxiliary.QueryInfo object.

Also note how the method places the SQL statement text in the queryText property of QueryInfo before exiting. If the <tablePane> *showQuery* value is true, this table displays itself, plus the query that generated it.

Class Member

```
Method CreateRS(Output tSC As %Status,
                pInfo As %ZEN.Auxiliary.QueryInfo) As %ResultSet
{
    Set tRS = ""
    Set tSC = $$$OK

    Set tSELECT = "ID,Name,Title"
    Set tFROM = "MyApp.Employee"
    Set tORDERBY = pInfo.sortColumn
    Set tSORTORDER = pInfo.sortOrder
    Set tWHERE = ""

    // Build WHERE clause based on filters
    If ($GET(pInfo.filters("Name"))'="") {
        Set tWHERE = tWHERE _ $SELECT(tWHERE="":"",1:" AND ") _
                "Name %STARTSWITH '" _ pInfo.filters("Name") _ "'"
    }
    If ($GET(pInfo.filters("Title"))'="") {
        Set tWHERE = tWHERE _ $SELECT(tWHERE="":"",1:" AND ") _
                "Title %STARTSWITH '" _ pInfo.filters("Title") _ "'"
    }

    Set sql = "SELECT " _ tSELECT _ " FROM " _ tFROM
    Set:tWHERE'="" sql = sql _ " WHERE " _tWHERE
    Set:tORDERBY'="" sql =
       sql _ " ORDER BY " _tORDERBY _ $SELECT(tSORTORDER="desc":" desc",1:"")

    Set tRS = ##class(%ResultSet).%New()
    Set tSC = tRS.Prepare(sql)
    Set pInfo.queryText = sql

    Quit tRS
}
```

1.2.4.4 QueryInfo Properties

The following table describes the properties on the %ZEN.Auxiliary.QueryInfo object that appears in the signature for both callback methods.

Like tablePane, the dataCombo, dataListBox, and repeatingGroup components can also use callbacks to generate a component from a %ResultSet. Some of the properties in the QueryInfo object apply only to tablePane queries. dataCombo, dataListBox, and repeatingGroup ignore any table-only properties, including those for columns, filters, and sorting.

Only tablePane and dataCombo support the queryText property.

Table 1–2: QueryInfo Properties

Property	Description
columnExpression	The *colExpression* values from each <column> in the <tablePane>. columnExpression organizes these values as a multidimensional array subscripted by <column> *colName*.
columns	The *colName* values from each <column> in the <tablePane>. columns organizes these values as a multidimensional array subscripted by column number (1–based).
filterOps	The *filterOp* values from each <column> in the <tablePane>. filterOps organizes these values as a multidimensional array subscripted by <column> *colName*.
filters	The *filterValue* values from each <column> in the <tablePane>. filters organizes these values as a multidimensional array subscripted by <column> *colName*.
filterTypes	The *filterType* values from each <column> in the <tablePane>. filterTypes organizes these values as a multidimensional array subscripted by <column> *colName*.
groupByClause	The *groupByClause* value for the <tablePane>, if supplied.
orderByClause	The *orderByClause* value for the <tablePane>, if supplied.
parms	Multidimensional array, subscripted by parameter number (1–based). This array contains any input values provided by <parameter> elements within the <tablePane>.
queryExecuted	Set this property to true in the method identified by *OnCreateResultSet*, to indicate that the newly created %ResultSet has already been executed and you do not want the method identified by *OnExecuteResultSet* to be called. The default is false.
queryText	The method identified by *OnCreateResultSet* can set this value to the actual query text, to be displayed if the *showQuery* value is true.
rowCount	If this value is set by the callback, upon return the rowCount property contains the number of rows returned by the query. After the query is executed, rowCount could be different from rows. Note that rowCount is a string, and not numeric, as its value might be `""` or `"100+"`. Any number of rows greater than 100 is represented as `"100+"`. When testing rowCount from JavaScript, if you want to convert to a numeric value use parseInt for base 10: `rowCount = parseInt(rowCount,10);`
rows	The number of rows requested. For tables, this is the *maxRows* value.
sortColumn	The *colName* of the current sort column selected by the user.
sortOrder	The table's current sort order (usually determined by user clicks) such as "asc" or "desc".
tableName	The <tablePane> *tableName* value.
whereClause	The *whereClause* value for the <tablePane>, if supplied.

1.2.5 Changing the Data Source Programmatically

You can change the data source for a <tablePane> at runtime in a variety of ways. The principle at work here is that the data source is on the server, so if you want to change the data source for a table based on user actions on the client side, you must work your way back to the server, as this topic shows.

In the following example, *queryClass* and *queryName* were used to define the data source, so we need to change them to new values on the server side. This example uses a JavaScript method in step 2 and a ZenMethod in step 3. Using an intervening JavaScript method as in step 2 is convenient if there are other actions you need to perform on the client side while changing the data source. Alternatively, the *onclick* from step 1 could invoke the ZenMethod directly, bypassing step 2:

1. There is a component on the Zen page whose response to *onclick* (or to some other user action) is to invoke a client side JavaScript method. For example:

   ```
   <button caption="Display Form" onclick="zenPage.setUpContextForm()" />
   ```

2. The client side JavaScript method invokes a server side ZenMethod that manipulates data source properties of the <tablePane>. For example:

   Class Member

   ```
    ClientMethod setUpContextForm() [ Language = javascript ]
   {
     this.SetQueryClassAndName("LTD.DomainModel.ContextList","GetAll")
     ctrl = this.getComponentById('ctrlList')
     ctrl.setModelClass('LTD.DomainModel.ContextList',this.getCurrentListId())
     zenSetProp('ContextId', 'hidden', 0)
     zenSetProp('ContextType', 'hidden',0)
   }
   ```

3. The server side ZenMethod gets the <tablePane> component and set its data source properties. For example:

   Class Member

   ```
   Method SetQueryClassAndName(queryClass As %String,
                               queryName as %String)
                               As %Status [ ZenMethod ]
   {
     Set obj=%page.%GetComponentById("listTable")
     Set obj.queryClass = queryClass
     Set obj.queryName = queryName
     Quit $$$OK
   }
   ```

1.3 Query Parameters

When you are working with SQL queries to generate the data for a Zen table, you sometimes need to provide values for query input parameters, defined as ? characters within the query. To do this, use <parameter> elements within the <tablePane> element. <radioSet>, <select>, <dataListBox>, <dataCombo>, <repeatingGroup>, and <multiSelectSet> can also contain <parameter> elements to support queries.

Attribute	Description
value	Specifies the parameter value: `<parameter value="Here is my value!"/>` The *value* supplied for a parameter can be a literal string, or it can contain a Zen #()# runtime expression.

When you supply a query directly, as with a *sql* attribute, each <parameter> element substitutes for one ? in the query syntax, in order from left to right, even if the values are the same. For example:

XML

```
<tablePane id="table"
           sql="SELECT ID,Name FROM MyApp.Employees
                WHERE Name %STARTSWITH ? AND
                ((Salary &lt; ?) OR (TotalCompensation &lt; ?))
                ORDER BY Name"
           >
     <parameter value="Z"/>
     <parameter value="100000"/>
     <parameter value="100000"/>
</tablePane>
```

The "Data Sources" section in this chapter provides several other examples of how to use the <parameter> element, including the following class query example. Here each <parameter> element substitutes for one argument in the class query:

XML

```
<tablePane id="table"
           queryClass="MyApp.Employee"
           queryName="ListEmployees">
    <parameter value="Sales"/>
    <parameter value="NEW YORK"/>
</tablePane>
```

When you work with %ZEN.Component.tablePane programmatically, If you are using one of the classes <radioSet>, <select>, or <multiSelectSet>, which do not implement the **setProperty**, you must first set an `id` for the parameter:

XML

```
<parameter value="Sales" id="param1"/>
```

The following example changes the value of the first parameter to Finance, re-executes the query on the server, and updates the contents of the tablePane to display the new results:

Class Member

```
ClientMethod changeParams() [ Language = javascript ]
{
    // find the tablePane component
    var table = zenPage.getComponentById('table');
    var param1 = zenPage.getComponentById("p1");
    param1.value='Finance';
    table.executeQuery();
}
```

1.4 Table Columns

The "Data Sources" section in this chapter explains that a <tablePane> draws an HTML table based on an SQL query. The table displays each row in the query resultset as a table row. A <tablePane> may also contain one or more <column> elements. <column> elements select which of the columns in the query resultset to display, and specify layout, style, and behavior for each column.

<tablePane> has a *showRowSelector* attribute which is true by default. If *showRowSelector* is true, the table displays an extra column at far left. This column appears empty when the table first displays. The purpose of this column is to indicate which rows are selected when the user selects them. If you want to suppress this column in your <tablePane>, set *showRowSelector* to false.

The "Generating an SQL Query" section includes the following example of how <column> elements can generate a table:

XML

```
<tablePane id="table"
    tableName="MyApp.Employee">
    <column colName="ID" />
    <column colName="Name"/>
    <column colName="Title" style="color: blue;"/>
</tablePane>
```

The <column> element is the XML projection of the %ZEN.Auxiliary.column class. <column> supports the general-purpose attributes described in the following table.

Attribute	Description
cellTitle	Text specifying the tooltip message for any cell within the column. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
colExpression	If the table is automatically constructing an SQL query, this is the SQL expression used to get the value of this column. For example: `colExpression="Customer->Name"`. Does not support aliasing. When using *colExpression*, you must also specify *colName*.
colName	The name of the SQL data column that this column is associated with. For more information, see colName.
disableSort	If true, disables sorting this column when the user clicks on the column header. If false, enables sorting. The default is for column sorting to be enabled in each column as long as the useSnapshot attribute for the containing <tablePane> is set to true.
header	Text specifying the column header. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types." The *header* value can be a literal string, or it can contain a Zen #()# runtime expression.
hidden	If true, this column is not be displayed. The default is false. *hidden* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types." The *hidden* value can be a literal string, or it can contain a Zen #()# runtime expression.
OnDrawCell	Name of a server-side callback method in the Zen page class. For more information see OnDrawCell.
seed	Allows you to pass some arbitrary value to the *OnDrawCell* callback.
style	CSS style value to be applied to the cells (HTML <td> elements) within this column. For example: `"color: red;"` The *style* value can be a literal string, or it can contain a Zen #()# runtime expression.

Attribute	Description
title	Text specifying the tooltip displayed when the user moves the mouse over the column header. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
width	Usually, the HTML *style* value for Zen tables is "fixed". This means that each column has a specific *width* value in the generated HTML page. Zen determines these values as follows: • You can specify a *width* value for any column. • If you do not specify a *width* for some columns, Zen assigns a *width* that is proportional to the size of the contents of that column (relative to other columns in the table). • If you do not supply a *width* value for any column in the table, Zen uses an HTML *style* value of "auto" for the entire table. The *width* value can be a literal string, or it can contain a Zen #()# runtime expression.

<column> also provides attributes that support dynamic filtering and linking for Zen table columns. Later topics describe these special-purpose <column> attributes:

- "Column Filters"
- "Column Links"

Finally, if you wish column layout to respond dynamically to user selections, the <tablePane> provides attributes that facilitate this for all columns in the table. See these sections:

- "Sorting Tables"
- "Selecting Rows and Columns"

When you work with %ZEN.Component.tablePane programmatically, you work with <column> elements as members of the tablePane columns property, a list collection of %ZEN.Auxiliary.column objects. Each <column> in the <tablePane> becomes a member of the columns collection in tablePane, associated with its ordinal position: 1, 2, 3, etc.

1.4.1 colName

The *colName* attribute provides the name of the SQL data column that this column is associated with. The *colName* value can be a literal string, or it can contain a Zen #()# runtime expression.

If any *colName* values in the <tablePane> are duplicates, the second column displays as "(duplicate) *colName*" to indicate that unless the second column is renamed, the <tablePane> may display unexpected behavior.

If no *colName* is specified for a column, the column is displayed without a data value. Typically this technique is used to display a link action in a row.

1.4.2 OnDrawCell

The *OnDrawCell* attribute provides the name of a server-side callback method in the Zen page class. This method injects HTML content into cells in the column using &html<> syntax or WRITE commands. Zen invokes this method whenever it draws the column, automatically passing it the following parameters:

- A pointer to the <tablePane> object.
- A string that gives the name of the SQL data column that this <column> is associated with.
- The *seed* attribute value from the <column>.

The callback must return a %Status data type. The following is a valid method signature:

Class Member

```
Method DrawYesNo(pTable As %ZEN.Component.tablePane,
     pName As %String,
     pSeed As %String) As %Status
{ }
```

To use the above method as the callback, the developer would set `OnDrawCell="DrawYesNo"` in the <column> statement:

XML

```
<column colName="WorkDone" header="Complete?"
     OnDrawCell="DrawYesNo"  />
```

The following OnDrawCell callback method interprets a Boolean value (1 or 0) to display the string Yes or No in the <tablePane> column.

Class Member

```
Method DrawYesNo(pTable As %ZEN.Component.tablePane,
     pName As %String,
     pSeed As %String) As %Status
{
  If %query(pName)
  {
    Write $$$Text("Yes")
  }
  Else
  {
    Write $$$Text("No")
  }
  Quit $$$OK
}
```

To retrieve the data value from the SQL column while inside your OnDrawCell callback method, use the `%query` function as shown in the code example above. `%query` is a function that takes one argument, a string that identifies the name of the SQL column. The signature of your OnDrawCell method provides this value automatically in the input parameter *pName*. Thus, the expression `%query(pName)` in your method resolves to the value contained in the SQL column that corresponds to this <tablePane> <column>.

The example tests to see if the expression `%query(pName)` is non-zero. If so, it places the word Yes in the <tablePane> column; otherwise it places the word No in the <tablePane> column.

Using the `%query` function in an OnDrawCell callback method is not the same as using the `%query` special variable in Zen runtime expressions. For information and examples using the `%query` special variable, with dot syntax, see the sections "Zen Special Variables" and "Zen Runtime Expressions" in the book *Developing Zen Applications*.

1.5 Table Style

<tablePane> offers the following attributes to control the general style of the table.

Attribute	Description
caption	Text specifying the caption to display for this table. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
extraColumnWidth	The HTML width to allow for extra columns, such as when multiple rows are selected or row numbers are displayed in the tablePane. The default width for an extra column is 30.
fixedHeaders	If true, the header of the table stays in position when the body of the table scrolls. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
bodyHeight	If *fixedHeaders* is true, *bodyHeight* provides an HTML length value that specifies the height of the body section of the table. The default *bodyHeight* is `"20.0em"`.
nowrap	If true, table cells disallow word wrapping. If false, they allow it. The default is true. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showRowNumbers	If true, display a row number column on the left-side of the tablePane. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showValueInTooltip	If true, the tooltip (HTML *title* attribute) displayed for cells within the table consists of the current value of the cell. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showZebra	If true, use zebra striping (alternating dark and light rows) to display the tablePane. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

1.6 Conditional Style for Rows or Cells

A <tablePane> may contain one or more <condition> elements. Each <condition> is a simple expression, based on the values of a given row, that controls the style of the row or of an individual cell within the row. For example:

XML

```
<tablePane id="table"
    sql="SELECT Name,Home_State FROM MyApp.Employee">
    <condition colName="Name"
               predicate="STARTSWITH"
               value="A"
               rowStyle="background: plum;"/>
</tablePane>
```

In the above example, every row in which the value of the Name column starts with “A” is displayed with a plum background.

Typically, the conditional style mechanism is used to highlight rows or cells containing special values (such as out-of-range or error cases). Adding conditions does increase the amount of processing needed to display a table, so use them sparingly.

The <condition> element supports the following attributes:

Attribute	Description
cellStyle	CSS style to be applied to cells within the target column, for rows in which this condition evaluates true. For example: `"color: red;"`
colName	Required. The name of the column that provides the data value to be evaluated by the <condition>. *colName* can be a literal string, or it can contain a Zen #()# runtime expression.
predicate	The logical operator used to evaluate the condition. *predicate* may be one of the following comparison operators: `""`, `"GT"`, `"EQ"`, `"LT"`, `"NEQ"`, `"GTEQ"`, `"LTEQ"`, `"EXTEQ"`, `"STARTSWITH"`, or `"CONTAINS"`. The default *predicate* is `"EQ"`. For details about each operator, see the “<condition> predicate Values” table, below.
rowStyle	CSS style to apply to rows in which this condition evaluates to true. For example: `"font-weight: bold;"`
targetCol	The name of the column that *cellStyle* applies to. If not specified, *colName* is used. If the target column displays a link, as discussed in the section Column Links, the *targetCol* must match the *linkCaption* attribute of the <column>. *targetCol* can be a literal string, or it can contain a Zen #()# runtime expression.
value	The literal value to be compared against the value in the column identified in *colName*. If enclosed within {} (for example, `"{Title}"`) *value* is treated as the name of another column, and the value in that column is used. *value* can be a literal string, or it can contain a Zen #()# runtime expression.

When a table is displayed, all <condition> elements within the <tablePane> are evaluated individually for each row in the table. If a <condition> evaluates true, then the *rowStyle* or *cellStyle* for the condition is applied to the row or cell, respectively.

The <condition> *predicate* attribute may have the following values.

Table 1–3: <condition> predicate Values

Predicate	Description
CONTAINS	True if the value in the column identified by *colName* contains (as a substring) the value specified by *value*.
EQ	True if the value in the column identified by *colName* is equal to the value specified by *value*.
EXTEQ	True if the filename in the column identified by *colName* has the file extension specified by *value*.
GT	True if the value in the column identified by *colName* is greater than the value specified by *value*.

Predicate	Description
GTEQ	True if the value in the column identified by *colName* is greater than or equal to the value specified by *value*.
LT	True if the value in the column identified by *colName* is less than the value specified by *value*.
LTEQ	True if the value in the column identified by *colName* is less than or equal to the value specified by *value*.
NEQ	True if the value in the column identified by *colName* is not equal to the value specified by *value*.
STARTSWITH	True if the value in the column identified by *colName* starts with the value specified by *value*.

When you work with %ZEN.Component.tablePane programmatically, you work with <condition> elements as members of the tablePane conditions property, a list collection of %ZEN.Auxiliary.condition objects. Each <condition> in the <tablePane> becomes a member of the conditions collection in tablePane, associated with its ordinal position: 1, 2, 3, etc.

1.7 Snapshot Mode

A <tablePane> can operate in *snapshot* mode. In this mode, Zen runs the table query once and copies these results to a temporary location on the server. Subsequent screen refresh operations display data from this temporary location, rather than resubmitting the query. Zen automatically manages the creation and lifecycle of the temporary snapshot data. Snapshot mode is particularly useful for working with multipage tables. Note that refreshRequired has no effect when you are using snapshot mode.

Important: There is a limit on the size of the data values that the temporary snapshot data structure can hold. For this reason, no single data value in a column can contain more than *n* characters. In turn, this means that no data value in the column can have its MAXLEN set to a value greater than *n*. Otherwise, a <SUBSCRIPT> error is generated when the user tries to sort the column. The value of *n* depends on the character set being used. *n* is the maximum length of a global subscript string in ObjectScript. This length depends on the character set; for English the limit is 508 and for Japanese it is less than 200. For details, see the "Determining the Maximum Length of a Subscript" section of *Using Caché Globals*.

The <tablePane> element supports the following attributes for snapshot mode.

Attribute	Description
useSnapshot	When true, this <tablePane> is in snapshot mode. This means that whenever data is fetched, it is copied into a server-side temporary location. Paging and sorting operations use this snapshot data and do not re-execute the query. If you want the user to be able to sort table columns by clicking on column headers, you must set *useSnapshot* to true. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types".
pageSize	For snapshot tables, this attribute specifies that you wish to display the data as multiple pages, and what the page size should be. 0, the default, means show all data on first page. This can only be set to a non-zero value when the table is in snapshot mode. Compare *maxRows*, which is the total number of rows to fetch in the <tablePane> query.

Any of the query mechanisms described in this chapter can be used with snapshot or direct (non-snapshot) mode. The following example specifies an SQL query to be used in snapshot mode:

XML

```
<tablePane id="table"
    sql="SELECT Name,Home_State FROM MyApp.Employee"
    useSnapshot="true"
    pageSize="25"
/>
```

The following %ZEN.Component.tablePane properties are not available as XML attributes in the <tablePane> definition, but they can be useful for working with snapshot tables once the page has been created.

Attribute	Description
clearSnapshot	A runtime flag that the client can set to true to force re-execution of the table query when Zen would otherwise use the stored snapshot. The default is false.
currPage	For snapshot tables with multiple pages of data, this is the (1–based) index number of the currently displayed page of data.

You can also programmatically adjust the values for *useSnapshot* and *pageSize* that were originally set by <tablePane> in XData Contents.

1.7.1 Fetching Data From the Server

The %ZEN.Component.tablePane class offers a **getRowData** method for tables in snapshot mode only. **getRowData** fetches the data values for a given row (0-based) from the server-side snapshot data. This data is packaged into a JavaScript object whose properties correspond to the names of the columns in the snapshot table; type conversion is handled appropriately. For non-snapshot tables or out-of-range row numbers, **getRowData** returns null.

1.7.2 Navigating Snapshot Tables

Several different options permit users to navigate multipage tables. The <tableNavigator> and <tableNavigatorBar> components provide a basic navigation interface. <tableNavigatorBar> is particularly useful for managing multipage tables. For details, see the section "Navigation Buttons."

If you wish to undertake additional programming, the %ZEN.Component.tablePane class provides a client-side JavaScript API that an application can use to implement the desired paging interface. These methods work only when the tablePane is in snapshot mode. They include:

Method	Description
`getPageCount()`	Calculates and returns the current number of pages within the table.
`getProperty('currPage')`	Returns the page number (1–based) of the current page displayed by the table.
`getProperty('pageSize')`	Returns the current page size.
`getProperty('rowCount')`	Returns the total number of rows within the table. Note that rowCount is a string, and not numeric, as its value might be `""` or `"100+"`. Any number of rows greater than 100 is represented as `"100+"`. When testing rowCount from JavaScript, if you want to convert to a numeric value use parseInt for base 10: `rowCount = parseInt(rowCount,10);`
`setProperty('currPage',pageno)`	Changes the current page displayed by the table to *pageno*.
`setProperty('pageSize',rows)`	Changes the current page size used by the table to *rows*.

1.8 Column Filters

A Zen table can create a “filter” to place above the header for any column. A filter is a simple box with an input field where a user can enter one or more search criteria. When the user submits these changes, the query associated with the <tablePane> is re-executed using the new criteria. Zen updates the table and nothing else on the page changes.

Filtering works only if the <tablePane> is using an automatically generated SQL statement or an OnCreateResultSet callback, and the callback generates the appropriate WHERE logic to implement the data filtering. When your table uses a generated SQL query, your page class can gather what the user enters; format it appropriately into the %ZEN.Component.tablePane properties groupByClause, orderByClause, and whereClause; then re-execute the table query.

Important: If you do not provide a *colName* value with the <column> element that specifies the filter, Zen does not create the filter.

The <column> element offers the following attributes for filters.

Attribute	Description
filterEnum	If *filterType* is `"enum"`, *filterEnum* defines the set of enumerated values used by the filter as a comma-separated list. For example: `"red,green,blue"` The enumerated values are displayed within a combo box. The names supplied in the *filterEnum* list appear as selections in the combo box unless *filterEnumDisplay* is defined.

Attribute	Description
filterEnumDisplay	If *filterType* is "enum", and if *filterEnumDisplay* provides a comma-separated list of values, the combo box displays these values in place of the corresponding *filterEnum* values. The *filterEnumDisplay* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *filterEnumDisplay* string that was a comma-separated list in the original language must remain a comma-separated list.
filterLabel	If specified, this is a label to display for the filter control. If there is a multipart filter control (such as a *filterType* of "range"), then *filterLabel* is assumed to contain a comma-separated list of labels.
filterOp	If this column has a filter, *filterOp* is the name of the SQL operator that should be used in conjunction with the filter. Supported values are: "", "%STARTSWITH", "=", ">=", "<=", "<>", ">", "<", "[", "BETWEEN", "IN", "%CONTAINS", and "UP[".
filterQuery	If *filterType* is "query", *filterQuery* defines the SQL statement used to provide the set of values for a drop-down list. If the query has more than one column, the first column is used as the logical value (this is what is used in a search) and the second column is used as a display value.
filterTitle	Text specifying the tooltip displayed when the user moves the mouse over the filter control. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
filterType	Specifies that this column should display a search filter control and indicates what type of filter control to display. Possible *filterType* values are: • "text" - display a text box. • "date" - display a date using a popup calendar control. If *filterOp* is "BETWEEN", then 2 controls are displayed. The column can also specify a *maxDate* and *minDate*. • "datetime" - same as "date" except that a complete timestamp (date and time) is used. • "enum" - display a set of enumerated values in a combobox. The possible choices are specified by *filterEnum* and *filterEnumDisplay*. • "query" - display a set of values in a combobox. The contents of the combobox are provided by executing the query specified by *filterQuery*. • "custom" - display a custom filter using the server-side callback method specified by *OnDrawFilter*.

Attribute	Description
filterValue	Current value of the column filter for this column. Typically this acquires a value after the user enters a value within a filter control, but you can set the *filterValue* to define an initial value. The meaning of *filterValue* depends on *filterOp*. When *filterOp* is: • `"IN"`, *filterValue* is treated as a comma-separated list of IN clause values. • `"%CONTAINS"`, *filterValue* is treated as a comma-separated list of %CONTAINS clause values. • `"BETWEEN"`, *filterValue* is treated as a comma-separated list of two values used for the BETWEEN clause of the query. • `"UP["`, *filterValue* is converted to a case insensitive value. • Anything else, *filterValue* is treated as a single value.
maxDate	Specifies the maximum date available in the calendar selector if *filterType* is “date” or “datetime”.
minDate	Specifies the minimum date available in the calendar selector if *filterType* is “date” or “datetime”.
OnDrawFilter	Name of a server-side callback method in the Zen page class. This method injects HTML content into the filter for this column using &html<> syntax or WRITE commands. Additional information follows this table.

Zen invokes the *OnDrawFilter* method when it draws the column, but only if the value of *filterType* is `"custom"` at that time. Zen automatically passes the method the following parameters:

- %ZEN.Component.tablePane — the <tablePane> object
- %String — the *colName* value from the <column>
- %ZEN.Auxiliary.column — the <column> object

The callback must return a %Status data type. The following is a valid method signature:

Class Member

```
Method DrawFil(pTable As %ZEN.Component.tablePane,
               pName As %String,
               pColinfo As %ZEN.Auxiliary.column)
               As %Status

{ }
```

To use the above method as the callback, the developer would set `OnDrawFilter="DrawFil"` for the <column>.

The following sample <tablePane> generates an SQL statement that displays the Name and Department of employees:

XML

```
<tablePane id="table"
           useSnapshot="true"
           tableName="MyApp.Employee">
    <column colName="ID" hidden="true"/>
    <column colName="Name" filterType="text" />
    <column colName="Department"
            filterType="enum"
            filterEnum="Sales,Accounting,Marketing"
            filterOp="=" />
</tablePane>
```

This example uses the <column> *filterType* attribute to specify that the Name column should display a column filter (“text” indicates that this filter displays a box in which the user may type text). If the user enters a value in the box (such as “A”) and presses **Enter**, the table is updated to only show rows where the Name column starts with “A”. (If the <column> does not specify a *filterOp* value, the default matching operation is %STARTSWITH.)

For the Department column the example displays a more sophisticated filter: a combo box showing 3 possible values. To do this, it sets the <column> *filterType* to “enum” and sets *filterEnum* to a comma-separated list of possible values. It also specifies that an exact match is required, by setting the *filterOp* value to "=".

A <tablePane> has filters active and enabled by default. You do not need to supply any <tablePane> attributes to enable filtering. However, should you want to override the default settings, <tablePane> offers the following attributes that control filtering for the table as a whole, not just for individual columns.

Attribute	Description
autoExecute	If true, this attribute causes the table query to be executed whenever a filter value is changed. *autoExecute* is "true" by default. When false, a page must explicitly cause the <tablePane> to run its query by calling its *executeQuery* method. This attribute has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”
filtersDisabled	If true, disable column filters (if any). When true, column filters are still displayed, but they are inactive. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”
headerLayout	Controls how to display the table header when column filters are used. Possible values are: • `"filtersOnTop"` — Display column filters above column headers. This is the default. • `"headersOnTop"` — Display column headers above column filters.
showFilters	If true, display column filters (if any) above the column headers. If false, do not display filters. The default is true. This attribute has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”

1.9 Column Links

Columns within a Zen table can display links, such as a link that takes the user to another page to edit the details of the object displayed within the current row. This link can either be displayed within a column that contains a data value (in this case, the data value is displayed as a link), or as an extra column in the table that contains the link.

The <column> element offers the following attributes for links.

Attribute	Description

link	If specified, this column is displayed as a link using the value of the *link* property as a URI. If you want to invoke a client-side JavaScript method in the link, start the URI with `javascript:` as in:
	`link="javascript:zenPage.myMethod();"`
	For more about this convention, see the example following the table.
	Alternatively, set *link* to # and use the *onclick* event to determine the action when the user clicks on this column. Doing this causes the *linkCaption* text to be formatted using whatever link styles are assigned by your CSS stylesheet:
	`<column onclick="return zenPage.test('#(%query.Name)#');" linkCaption="Test" link="#"/>`

linkCaption	If this column has an action defined (*link* or *onclick*) and does not display a data value, the *linkCaption* specifies the text to use as the caption for the link. If you are using a conditional style, as described in the section Conditional Style for Rows or Cells, *linkCaption* must match the *targetCol* attribute of the <condition> element that supplies the style. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
linkConfirm	If specified and this column has a *link* defined, the *linkConfirm* text is displayed as a confirmation message before the link is executed. If there is an *onclick* action defined for this column, then *linkConfirm* is ignored. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
onclick	The event handler that runs each time the user clicks the mouse on a cell in this column. See "Zen Component Event Handlers." If the column does not have data associated with it then you must set the *linkCaption* property so that the user has text to click. When you use onclick, set *link* to # as described for *link* in this table.

The following is an example of using JavaScript code to further process the <column> *link* value before using it to go to another page:

XML

```
<tablePane id="table"
    sql="SELECT TOP 100 ID,Item,Price FROM MyApp.Inventory ORDER BY Price">
    <column colName="ID" hidden="true"/>
    <column colName="Item"
            link="MyApp.ItemEdit.cls?ID=#(%query.ID)#"
            linkCaption="View information on this item."
    />
    <column link="javascript:zenPage.addToCart('#(%query.ID)#');"
            linkCaption="Add to cart"
            linkConfirm="Do you wish to add this item to your cart?"
    />
</tablePane>
```

This example does the following:

1. The <tablePane> *sql* value specifies that this table displays information about the top 100 items in the MyApp.Inventory table ordered by price.
2. The <tablePane> needs the value of the “ID” column (as part of the links) but there is no need to display this value. Therefore, the <column> *hidden* value is true.
3. The <tablePane> needs the “Item” column to contain a link to a page that displays information about a specific item. Therefore, the <column> *link* value defines this link. The <column> *linkCaption* value defines a tooltip message that is displayed when the user moves the mouse over this link.
4. The <tablePane> defines an extra column (with no data displayed in it) that contains an “Add to cart” link. This link invokes a client-side **addToCart** method when the user clicks on it. The *linkConfirm* property specifies a confirmation message displayed to the user before executing the link.

The two *link* values used within this <tablePane> example make use of the Zen expression syntax to include the value of row-specific data within the link. See the section “Zen Runtime Expressions” in the “Zen Pages” chapter of *Developing Zen Applications*. The following expression refers to the ID column within the current row of the table:

```
#(%query.ID)#
```

For the second of the two *link* examples to work, the client-side **addToCart** method must compose the URI and invoke the new page. Using a client-side method is important when you need to encode special characters that may appear in the text value returned by the query; for example:

Class Member

```
ClientMethod addToCart(identifier) [Language = javascript]
{
  var page = "MyApp.AddToCart.cls?ID=" + encodeURIComponent(identifier);
  this.gotoPage(page);
  return;
}
```

1.10 User Interactions

Zen tables provide built-in mechanisms to support basic user interactions such as navigating pages, sorting columns, and selecting rows in a table.

1.10.1 Navigation Buttons

The <tableNavigator> component automatically displays a set of buttons for moving through the pages of a <tablePane>. The <tableNavigatorBar> has identical syntax, but displays extra buttons to help users navigate large, multipage tables.

To use either component, place it anywhere on the same page as a <tablePane> and set its *tablePaneId* attribute value to match the *id* value from the <tablePane>. For example:

XML

```
<pane id="tPane">
  <tablePane id="myTable"
             tableName="MyApp.Employee">
    <column colName="ID" hidden="true"/>
    <column colName="Name"/>
  </tablePane>
  <tableNavigatorBar tablePaneId="myTable" />
</pane>
```

If a <tableNavigator> or <tableNavigatorBar> is placed within a composite element, the corresponding <tablePane> must be placed within the same composite element.

1.10.2 Navigation Keys

The <tablePane> can specify event handling for user key clicks as follows.

Property	Description
onkeypress	Specifies the event handler for events generated when the user presses the keys **Up**, **Down**, **Page Up**, **Page Down**, **Home**, **End** while focus is in the table. See "Zen Component Event Handlers." The attribute *useKeys* must be true, so that the tablePane captures the user keystrokes.
useKeys	If true, this tablePane captures user keystrokes **Up**, **Down**, **Page Up**, **Page Down**, **Home**, **End**, and uses them for simple table navigation. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

1.10.3 Sorting Tables

The <tablePane> must use snapshot mode to allow users to sort table columns. If you want the user to be able to sort table columns by clicking on their headers, you must set the <tablePane> *useSnapshot* attribute to true; by default, *useSnapshot* is false. For details and important limitations, see the "Snapshot Mode" section.

When *useSnapshot* is true, you can selectively disable sorting for a column by setting the <column> *disableSort* attribute to true. For more about column attributes like *disableSort* see the "Table Columns" section.

When snapshots are enabled, the user can sort the table according to a particular column simply by clicking on a column header: First click, ascending order; second click, descending order; third click, unsorted; and so on. You can set an initial value for *sortOrder* when you add the <tablePane> to the page, as described in the following table. However, *sortOrder* usually takes its value based on user actions (clicking column headers in the table).

Property	Description
sortOrder	When the user clicks on a column header, Zen sorts the table based on the values in that column and the order specified by the *sortOrder* value. *sortOrder* toggles between its possible values `"asc"` (sort in ascending order) and `"desc"` (sort in descending order) each time the user clicks in that column header. *sortOrder* does not affect the query itself (it does not interact with the query ORDER BY setting). *sortOrder* simply controls the order in which the table displays the resultset returned by the query. The *sortOrder* value can be a literal string, or it can contain a Zen #()# runtime expression.

For columns that contain date data, clicking on the column header sorts the dates in correct chronological order only if the display format for the dates is either the default format dictated by the current locale setting, or ODBC format. Other formats are sorted in alphabetical order.

For more information on date formats, see the description of *dformat* in the "Parameters" section for the $ZDATETIME ($ZDT) function in the *Caché ObjectScript Reference*.

1.10.4 Selecting Rows and Columns

The user can select rows or columns within a Zen table by clicking on them. Zen indicates the current row visually and notifies the application of the event by triggering the event handler provided by the <tablePane> *onselectrow* attribute.

%ZEN.Component.tablePane supports a number of properties for row and column selection. Many of these properties can be set as <tablePane> attributes, but some of them can only take their values at runtime in response to user actions.

Property	Description
currColumn	The *colName* of the column most recently selected by the user. You may allow user actions to provide a value for this property, or you can set it. The *currColumn* value can be a literal string, or it can contain a Zen #()# runtime expression.
multiSelect	If true, the user can select multiple rows within the table. Zen displays an extra column, containing check boxes, to indicate which rows are selected. The default is false. *multiSelect* and *rowSelect* can be true or false independently of each other.
ondblclick	The event handler for events generated when the user double-clicks on a row in this table. See "Zen Component Event Handlers."
onheaderClick	Client-side JavaScript expression that runs whenever the user clicks on a column header within this table. Zen stores the name of this column in the *currColumn* property.
onmultiselect	If *multiSelect* is true, this is the client-side JavaScript expression that runs whenever the user changes the set of multiply selected rows within this table.
onselectrow	If *rowSelect* is true, this is the client-side JavaScript expression that runs whenever the user selects a new row within this table. This happens only if *showRowSelector* is true.
rowSelect	If true, the user can select a row within the table (one row at a time). The default is true. *multiSelect* and *rowSelect* can be true or false independently of each other.

Property	Description
selectedIndex	The (0–based) index number of the currently selected row. This value is relevant only if the *showRowSelector* property is true. For snapshot tables, this is row number within the current page.

selectedRows	Read-only.
	When the <tablePane> has *multiSelect* set to true and a *valueColumn* defined, its *selectedRows* string indicates which rows are currently selected in the table. The string does this by providing a comma-separated list of the *valueColumn* values in each selected row. Consecutive commas in the string indicate that the row in that position is not selected.
	The *selectedRows* string looks like the following example, in which 3 of the 14 rows are selected:
	`",,,value,,,value,,,,,value,,"`
	If the <tablePane> has no *valueColumn* defined, but has *multiSelect* set to true, the *selectedRows* string provides no information about selected rows, and looks like this:
	`",,,,,,,,,,,,,"`

showRowSelector	If true, the table displays an extra column at far left. This column appears empty when the table first displays. The purpose of this column is to indicate which rows are selected when the user selects them. *showRowSelector* is true by default. If you want to suppress this extra column in your table, set *showRowSelector* to false.
value	This is the logical value used to determine which is the currently selected row. *value* works with *valueColumn*. The value may be empty (""). Do not access this value directly; use `getProperty('value')` instead.
valueColumn	A <tablePane> can have a logical *value* defined. Each time the table is refreshed, in each row Zen tests this logical *value* against the actual value that appears in the *valueColumn* in that row. Zen selects any row(s) that contain *value* in *valueColumn*. This implies the following: • You can preset the *value* of a <tablePane> and the row(s) that match are selected when the table is first displayed. • The current selection is preserved when you sort the rows in a table.

1.11 Table Refresh

When you refresh a table, only the table refreshes. It is not necessary for the entire Zen page to refresh itself in order to refresh a table.

Typically, Zen refreshes the visible contents of the table automatically as needed. When you work with tables programmatically, you can also explicitly refresh table contents. The techniques are as follows:

- Call the **%ResetQuery** method of the %ZEN.Component.tablePane from a server-side method. This forces the tablePane to re-execute its query.

- Call the **executeQuery** method of the %ZEN.Component.tablePane. This causes the data query to be re-executed and updates the visible contents of the table to reflect the current values of tablePane properties.

Note that calling **%ResetQuery** has no effect when you are using snapshot mode.

1.12 Table Touchups

Any time you set an attribute value for <tablePane> in XData Contents, the corresponding property in the tablePane object automatically acquires this value. This might be just enough programming for your purposes. Nevetheless, %ZEN.Component.tablePane offers many more opportunities for programmatic interaction on the client or server sides.

1.12.1 Data Values

If you wish to touch up the data values displayed in the table, you have these options:

- Touching up values set in XData Contents, just prior to display, by setting tablePane properties in the **%OnAfterCreatePage** callback of the page class.
- Examining read-only properties of %ZEN.Component.tablePane such as lastUpdate and rowCount to determine the current state of the table. Note that rowCount is a string, and not numeric, as its value might be `""` or `"100+"`. Any number of rows greater than 100 is represented as `"100+"`.
- Resetting values of certain tablePane properties and then refreshing the table.

1.12.2 Header and Body Alignment

When viewed in Internet Explorer with *fixedHeaders* set to true, <tablePane> headers and body columns may become misaligned. For this reason, the %ZEN.Component.tablePane class offers a client-side JavaScript method called **resizeHeaders** that checks for alignment issues in the rendered table and, if needed, reformats the header with padding to account for the space taken up by a vertical scrollbar in the body of the table.

resizeHeaders calculates the size of the padding based on the actual size of the scrollbar on the rendered page, and automatically accounts for the differences between Internet Explorer 6 and 7 as well as any of the minor variations in scrollbar width that crop up under some of the Windows desktop themes. You do not need to use **resizeHeaders** unless you actually observe problems in Internet Explorer when viewing <tablePane> components with *fixedHeaders* set to true. Then, if you see this problem, you can fix it as follows:

1. Implement the client-side JavaScript method **onresizeHandler** in the Zen page class that displays the <tablePane>. **onresizeHandler** must be implemented in the page class because the resize event is only supported for the current `zenPage` object and does not propagate to any components on the page. For a list of similar client-side callback methods. see the section "Client-Side Page Callback Methods" in the "Zen Pages" chapter of *Developing Zen Applications*.
2. Ensure that you **onresizeHandler** implementation explicitly calls the **resizeHeaders** method of the tablePane object.

2

Zen and SVG

Scalable Vector Graphics (SVG) is a language that allows you to describe two-dimensional vector graphics in XML format. The SVG language specification is available on the web site www.w3.org/TR/SVG/.

Zen uses SVG to display high-performance, data-driven charts and meters. You can use the built-in SVG components to define eye-catching corporate dashboards that update their statistics in real time. You can also define your own SVG components. An SVG component is any component that inherits from %ZEN.SVGComponent.svgComponent. These components render dynamic SVG images that change their appearance in response to data values.

Note: If you want to display a static SVG file on the Zen page, use <iframe>. If you want to use a static SVG file as the image for a button control, use <image>. The conventions described in the following topics apply to dynamic SVG components only.

This chapter describes how to place SVG components on the Zen page. Topics include:

- "SVG Component Layout"
- "SVG Component Attributes"
- "Meters"
- "Charts"
- "<radialNavigator>"
- "<ownerDraw>"

2.1 Fonts for SVG

If no font-family is specified, SVG graphics use whatever default font the SVG engine provides. If you wish, you can specify the SVG default font for your system by setting the global ^%ZEN.DefaultSVGFontFamily to the name of a font-family. This causes Zen to create a CSS font-family definition using that font name. Zen automatically applies this font-family to all SVG text elements unless they explicitly provide a value for a CSS font-family.

For example, the MingLiU font is useful when working in Japanese with Internet Explorer on Microsoft Vista:

```
Set ^%ZEN.DefaultSVGFontFamily="MingLiU"
```

2.2 SVG Component Layout

The following components allow you to place SVG components on a Zen page:

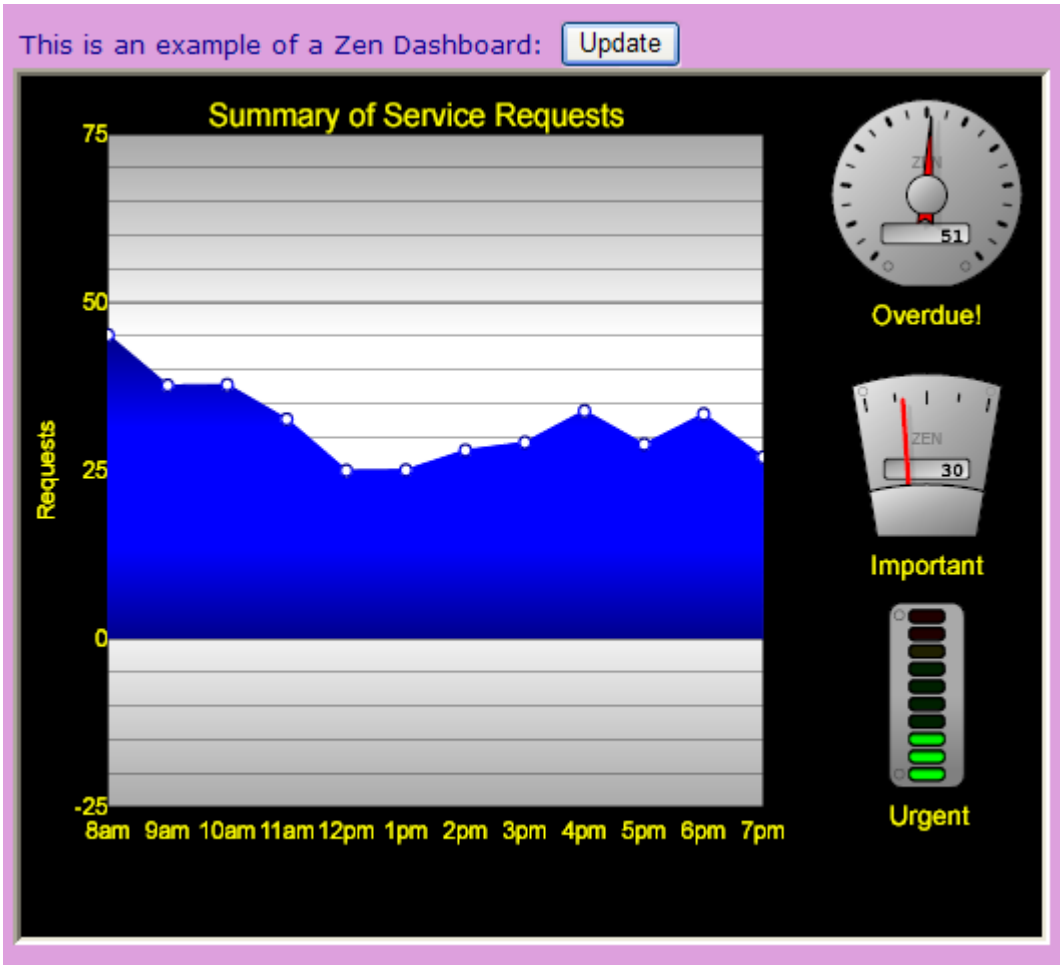
- "<svgFrame>"
- "<svgGroup>"
- "<svgSpacer>"

2.2.1 <svgFrame>

<svgFrame> is a Zen component that creates a rectangular frame on the Zen page, into which you can place SVG components. Only SVG components may appear inside this frame. Any dynamic SVG component, such as a meter or chart, requires an <svgFrame> to contain it.

<svgFrame> inherits from %ZEN.Component.component. This gives <svgFrame> the usual component style attributes — *height*, *width*, *label*, etc — as described in the "Zen Style" chapter of *Using Zen*. This convention also allows <svgFrame> to be placed within a <page>, or contained within an <hgroup> or <vgroup>, just like any other Zen component. For details, see the "Zen Layout" chapter of *Using Zen*.

The following figure shows a mix of Zen components and SVG components on a page. This is similar to the output provided by the sample class ZENDemo.Dashboard in the SAMPLES namespace.



The following XData Contents block contains the components that produced the above figure. These are a mix of:

- Zen components (<hgroup>,<vgroup>,<html>, <spacer>, <button>, <svgFrame>)
- SVG components (<speedometer>, <fuelGauge>, <lightBar>, <lineChart>, <svgGroup>, <svgSpacer>)

Class Member

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
<page cellStyle="background-color: plum;"
      xmlns="http://www.intersystems.com/zen">
  <hgroup>
    <spacer width="10"/>
      <vgroup valign="top">
        <spacer height="20"/>
        <hgroup>
          <html
            enclosingStyle="margin: 3px; font-size:1.2em; color: darkblue;">
            This is an example of a Zen Dashboard:
          </html>
          <spacer width="10" />
          <button caption="Update" onclick="zenPage.updateData();"/>
        </hgroup>
        <svgFrame id="svgFrame" width="600" height="430"
                  frameStyle="border-style: inset;"
                  backgroundStyle="fill: black;"
                  layout="horizontal">
          <svgSpacer width="20" />
          <svgGroup layout="vertical">
            <svgSpacer height="20" />
            <lineChart id="chart" width="400" height="420"
                       title="Summary of Service Requests"
                       backgroundStyle="fill: black;"
                       plotAreaStyle="fill: url(#glow-silver);"
                       lineStyle="stroke: black;" chartFilled="true"
                       seriesColors="url(#glow-blue)" seriesCount="1"
                       seriesSize="12" markersVisible="true"
                       marginRight="5" marginLeft="10"
                       ongetData="return zenPage.getChartData(series);"
                       ongetLabelX="return zenPage.getChartLabelX(value);">
              <yAxis id="yAxis" baseValue="0"
                     minValue="-25" maxValue="75"
                     majorUnits="25" minorUnits="5"
                     title="Requests" minorGridLines="true"/>
            </lineChart>
          </svgGroup>
          <svgGroup layout="vertical">
            <svgSpacer height="20" />
            <speedometer id="speed3" label="Overdue!" animate="true"
                         labelStyle="fill: yellow;"
                         lowLampColor="url(#glow-green)"
                         rangeUpper="100" width="125" height="125"/>
            <fuelGauge id="speed1" label="Important" animate="true"
                       labelStyle="fill: yellow;"
                       rangeUpper="100" width="125" height="125"/>
            <lightBar id="speed2" label="Urgent" animate="true"
                      labelStyle="fill: yellow;"
                      lowLampColor="url(#glow-green)"
                      rangeUpper="100" width="100" height="125"/>
          </svgGroup>
        </svgFrame>
      </vgroup>
    </hgroup>
  </page>
}
```

An <svgFrame> element may contain as many nested <svgGroup> and <svgSpacer> elements as are required to achieve the desired layout. An <svgFrame> may also contain <parameter> elements to support its *svgPage* attribute. For details, see the *svgPage* description in the following table.

<svgFrame> has the following attributes:

Attribute	Description
Zen component attributes	<svgFrame> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
backgroundStyle	SVG CSS style definition (Styles within SVG are CSS compliant, but there is a different set of styles available.) Specifies the background style for the frame. This style must include a fill value, or mouse events within this frame do not work correctly. The default *backgroundStyle* is: `"fill: white;"`
disabled	If true, this frame and its children are disabled. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
dragCanvas	If true, the user can use the pointing device (mouse) to drag the canvas of this frame. This updates the values of the *offsetX* and *offsetY* attributes and moves the shapes on the canvas. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
editMode	Edit mode for this frame. Possible values are: • "none" — The user cannot edit the contents of this frame. This is the default. • "select" — The user can click on an SVG component to select it. • "drag" — The user can click on an SVG component, hold down the mouse button, and drag the SVG component to a new position.
frameStyle	CSS style definition that applies to this frame. For example, to produce the recessed, beveled border shown in the illustration above, you would use: `frameStyle="border-style: inset;"`
gridX	If *snapToGrid* is true, *gridX* defines the HTML width of each cell in the sizing grid. The default is 25.
gridY	If *snapToGrid* is true, *gridY* defines the HTML height of each cell in the sizing grid. The default is 25.

Attribute	Description
layout	Specifies how the SVG components within this frame should be laid out. Possible values are: • "none" or the empty value "" — no layout is provided. When this is the case, components may be placed using specific coordinates *x* and *y*. See the *x* and *y* attributes in the section "SVG Component Attributes." • "vertical" — components within this group are laid out vertically. • "horizontal" — components within this group are laid out horizontally. • "flow" — components within this group are placed in rows. Items are placed horizontally until the width of the container is exceeded, and then components are placed on the next row.
offsetX	Offset, along the x-axis, of the coordinates of this frame from its upper, left-hand corner. The default is 0.
offsetY	Offset, along the y-axis, of the coordinates of this frame from its upper, left-hand corner. The default is 0.
ondragCanvas	The *ondragCanvas* event handler for the <svgFrame>. Zen invokes this handler each time the user drags the background canvas using a pointing device. See "Zen Component Event Handlers." A boolean variable, *done*, is passed to this event handler to indicate if the operation is complete.
onmouseWheel	Client-side JavaScript expression that runs whenever the user moves the mouse wheel over the background rectangle of this frame. Currently only available in FireFox.
onmoveItem	Client-side JavaScript expression that runs whenever this frame is in drag mode and the user moves one or more selected items. Refer to the *selectedItems* property for the list of items. A boolean variable, *done*, is passed to this event handler to indicate if the operation is complete.
onresizeItem	Client-side JavaScript expression that runs whenever this frame is in drag mode and the user resizes one or more selected items. Refer to the *selectedItems* property for the list of items. A boolean variable, *done*, is passed to this event handler to indicate if the operation is complete.
onselectItem	Client-side JavaScript expression that runs whenever the user changes the number of selected items in this frame (by selecting or unselecting an item). A variable, *item*, is passed to this event handler and refers to the item most recently selected or unselected.
onzoom	Client-side JavaScript expression that runs whenever the user changes the zoom level for this frame.
snapToGrid	If true, all mouse operations (sizing and dragging) are constrained to occur on the grid defined by *gridX* and *gridY*. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

Attribute	Description
svgAutoSize	Controls the size of the SVG drawing canvas within the frame. If *dragCanvas* is true, the *svgAutoSize* property is ignored. Otherwise, if *svgAutoSize* is true, Zen calculates (and updates) the canvas size automatically based on the frame contents, with the minimum canvas size being *svgWidth* by *svgHeight*. If *svgAutoSize* is false, the canvas size is determined by the values of *svgWidth* and *svgHeight*. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
svgHeight	Controls the size of the SVG drawing canvas within the frame. *svgHeight* gives the canvas height (using any valid SVG measurement units). If not specified, the default *svgHeight* is 100% of the *height* of the <svgFrame>. If *height* is also not specified, the default *height* (and *svgHeight*) is 100.

svgPage	Identifies a specialized CSP page that serves SVG content, defining styles and colors for SVG components in the frame. If provided, the *svgPage* value must be the name of a class that extends %ZEN.SVGComponent.svgPage. Otherwise, the frame uses the base class %ZEN.SVGComponent.svgPage by default.
	
	When you provide a value for the *svgPage* attribute, you can also include <parameter> elements within the <svgFrame>. Zen passes the *value* strings from these <parameter> elements to the *svgPage* class as URI parameters. For example:
	
	`<parameter value="I am a URI parameter!"/>`
	
	The *value* supplied for a <parameter> can be a literal string, or it can contain a Zen #()# runtime expression.
	
	Use the <parameter> attribute *paramName* to assign names to the parameters you use with <svgFrame>. For example:
	
	`<parameter paramName="First" value="I am a URI!"/>`

svgWidth	Controls the size of the SVG drawing canvas within the frame. *svgWidth* gives the canvas width (using any valid SVG measurement units). If not specified, the default *svgWidth* is 100% of the *width* of the <svgFrame>. If *width* is also not specified, the default width (and *svgWidth*) is 300.
zoom	Decimal value, at least 1.0 (the decimal portion of the number may be omitted). This value indicates the zoom factor for the frame. 100 means no zoom; this is the default. Values larger than 100 increase image size, while smaller values decrease it.
zoomLevels	Comma-separated list of suggested zoom values. The default is: `"10,25,50,75,100,125,150,175,200,300,400,500"`
zoomWithWheel	If true, this frame automatically zooms in and out in response to mouse wheel events. The default is false. Currently only available in FireFox. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

Important: It is *not possible* for <svgFrame> to contain any component that inherits from %ZEN.Component.component, such as <iframe>, <html>, <spacer>, <hgroup>, <vgroup>, <button>, <link>, or any of the other Zen components discussed in this book. <svgFrame> can contain SVG components only.

There are some cases where it is important to have programmatic access to the window object for the embedded SVG frame. The %ZEN.SVGComponent.svgFrame class has a client-side property called svgWindow that points to the SVG window object embedded within the <svgFrame> element

2.2.2 <svgGroup>

<svgGroup> is a special container designed to contain and lay out SVG components within <svgFrame>. <svgGroup> is not a Zen component (%ZEN.Component.component), nor is it a Zen group component (%ZEN.Component.group). <svgGroup> is an Zen SVG component (%ZEN.SVGComponent.svgComponent) with the ability to contain other Zen SVG components.

<svgGroup> has the following attributes:

Attribute	Description
SVG component attributes	<svgGroup> has the same general-purpose attributes as any SVG component. For descriptions, see the section "SVG Component Attributes."
disabled	If true, this group and its children are disabled (hidden). The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
layout	Specifies how the SVG components within this group should be laid out. Possible values are "horizontal" and "vertical". If *layout* is set to "none" or the empty value "", components may be placed using specific coordinates *x* and *y*. See the *x* and *y* attributes in the section "SVG Component Attributes."

2.2.3 <svgSpacer>

The <svgSpacer> element is useful within <svgGroup> containers. Use <spacer> with a *width* value to inject additional space in a horizontal <svgGroup>, or *height* for additional space within a vertical <svgGroup>.

<svgSpacer> has the same general-purpose attributes as any SVG component. For descriptions, see the section "SVG Component Attributes."

2.2.4 <rect>

The Zen <rect> element draws a simple rectangle. This is not the same as the <rect> element defined by the SVG language specification. Zen <rect> is a built-in Zen SVG component that you can place within an <svgFrame> or <svgGroup>.

The Zen <rect> element takes up space within an <svgFrame> or <svgGroup>, but in a different way than <svgSpacer>. The conceptual difference between <svgSpacer> and <rect> is that <rect> may have visible style attributes, such as a fill color.

Attribute	Description
SVG component attributes	<rect> has the same general-purpose attributes as any SVG component. For descriptions, see the section "SVG Component Attributes."
rx	Radius of curve for the corners (using any valid SVG measurement units).
style	SVG CSS style definition. (Styles within SVG are CSS compliant, but there is a different set of styles available.)

2.3 SVG Component Attributes

All SVG components have the following style attributes. These are entirely distinct from the attributes supported by ordinary Zen components.

Table 2–1: SVG Component Attributes

Attribute	Description
boundless	If true, this component is boundless. That is, its enclosing SVG element is a simple group <g> instead of the usual SVG element. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
height	Height of this component (using any valid SVG measurement units). The exact effect of setting this value depends on the component. In the case of built-in SVG components, the effect is usually straightforward. A *height* value larger than the *height* of the containing <svgFrame> causes the SVG component to appear cropped by the frame.
hidden	If true, this component is disabled (hidden). The default is false. Changing the hidden state of an SVG component causes the layout of the SVG frame to be recalculated. That is, if *layout* is not equal to "none" the other components move into the space that was occupied by the hidden component. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
id	Name that can be used to select the SVG component so that its attributes can be updated. For example, displays that use meters might periodically update the *value* that the meter represents.
name	Specifies a name for the component.
onclick	The *onclick* event handler for the SVG component. Zen invokes this handler each time the user clicks the mouse on this shape. See "Zen Component Event Handlers."
position	Used for placing a fixed controller over the rest of an SVG canvas. If the *position* of an SVG component is "fixed" this shape does not scroll with its canvas, nor can it be dragged with the mouse. If *position* is "relative" (the default) this shape scrolls, and can be dragged.
preserveAspectRatio	By default, Zen preserves the aspect ratio (height relative to width) for any SVG component when you change its size. If you set *preserveAspectRatio* to "none", Zen does not preserve the aspect ratio and allows your *height* and *width* changes to operate independently of each other.
viewBoxHeight	Height of the view box for this component (using any valid SVG measurement units). If a *viewBoxHeight* attribute is provided, its value is used as the height of the view box. Otherwise, the *height* value for the component is used.
viewBoxWidth	Width of the view box for this component (using any valid SVG measurement units). If a *viewBoxWidth* attribute is provided, its value is used as the width of the view box. Otherwise, the *width* value for the component is used.

Attribute	Description
width	Width of this component (using any valid SVG measurement units). The exact effect of setting this value depends on the component. In the case of built-in SVG components, the effect is usually straightforward. A *width* value larger than the *width* of the containing <svgFrame> causes the SVG component to appear cropped by the frame.
x	x position of this component (using any valid SVG measurement units). The actual position of the component may depend on the *layout* of its enclosing <svgGroup>. If the <svgGroup> has vertical or horizontal layout, this *x* coordinate is ignored. However, if `layout=""` this *x* coordinate takes effect. *x* is a positive value relative to an origin of (0,0) at the top left corner of the <svgGroup>.
y	*y* position of this component.

2.4 Meters

A *meter* is an SVG component that displays a graphical representation of a single numeric value. Each meter is a class derived from %ZEN.SVGComponent.meter that generates the SVG required to display itself.

Zen provides several built-in meters. To place one of them on the Zen page, provide the corresponding meter element within an <svgFrame> or <svgGroup>:

- "<fuelGauge>"
- "<indicatorLamp>"
- "<lightBar>"
- "<slider>"
- "<smiley>"
- "<speedometer>"
- "<trafficLight>"
- To create your own style of meter, see the "Custom Components" chapter in *Developing Zen Applications*.

This topic describes how to supply a meter with a value, lists the attributes that all meters share in common, then describes the unique attributes for each type of meter listed above.

2.4.1 Providing Data for Meters

Your code can dynamically update the value displayed by a meter in one of two ways:

- From a JavaScript method in the page class, set the meter's *value* attribute using the meter class methods **getComponentById** and **setValue** as follows:

  ```
  this.getComponentById("myMeterID").setValue(myNewValue);
  ```

  Where *myMeterID* matches the *id* attribute value for the meter, and *myNewValue* is a variable that contains a single, numeric value.

- Associate the meter with a data controller, as described in the chapter "Model View Controller." For example, if your XData Contents block contains an <dataController> reference that looks like this:

XML

```
<dataController id="source" modelClass="myPackage.MyModel" modelId="1"/>
```

And if the referenced *modelClass* myPackage.MyModel contains a property called Automobiles, then your XData Contents block can also contain a meter definition that looks like this:

XML

```
<trafficLight id="myLight" controllerId="source"
              height="150" width="75"
              dataBinding="Automobiles"
              label="Autos" labelStyle="fill: yellow;" />
```

This technique works when the meter's *controllerId* value matches the <dataController> *id* value, and the meter's *dataBinding* value matches the name of a property defined in the <dataController> *modelClass*.

2.4.2 Meter Attributes

All meters have the following attributes, which define their style and behavior.

Table 2–2: Meter Component Attributes

Attribute	Description
SVG component attributes	Meters have the same general-purpose attributes as any SVG component. For descriptions, see the section "SVG Component Attributes."
animate	Some of the built-in Zen meters provide animation. For example, if the meter uses a needle to indicate a value, animation causes the needle to "swing" from the previous value to the new value. If a meter supported animation, this attribute controls whether animation is on (true) or off (false). The default is "true". This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
controllerId	Identifies the data controller that provides the data for this meter. The *controllerId* value must match the *id* value provided for that <dataController> component. For details, see the chapter "Model View Controller."
dataBinding	If this meter is associated with a data controller, this attribute identifies the specific property within the <dataController> *modelClass* that provides the value for this control. For details, see the chapter "Model View Controller."
label	A text label for the meter. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
labelStyle	SVG CSS style definition. (Styles within SVG are CSS compliant, but there is a different set of styles available.) When Zen lays out this meter, it applies this style to the *label* text.
onnotifyView	The *onnotifyView* event handler for the meter. See "Zen Component Event Handlers." This attribute applies if the meter is associated with a data controller. Zen invokes this handler each time the data controller connected to this meter raises an event. For details, see the chapter "Model View Controller."

Attribute	Description
rangeLower	Integer or decimal value that defines the low end of the range for this meter. The default is 0.
rangeUpper	Integer or decimal value that defines the high end of the range for this meter. The default is 100.
scaleFactor	Integer or decimal factor used to scale the values provided to this meter. Zen scales the incoming values before comparing them with *rangeLower*, *rangeUpper*, and so on. The default is 1 (no scaling).
thresholdLower	Integer or decimal value that defines a threshold for meter behavior. The meter does something when the meter value falls below this value. Typically, the *thresholdLower* value is greater than *rangeLower* and serves as a warning that the meter value is approaching *rangeLower*. The default is 0.
thresholdUpper	Integer or decimal value that defines a threshold for meter behavior. The meter does something when the meter value rises above this value. Typically, the *thresholdUpper* value is less than *rangeUpper* and serves as a warning that the meter value is approaching *rangeUpper*. The default is 90.
value	The current integer or decimal value of the meter (actually stored as a string). Although you can set this value while placing the meter on the page, generally you do not do this. Instead, you use **getComponentById** and **setValue**, or associate the meter with a %ZEN.Auxiliary.dataController, as described in the paragraph prior to this table.

2.4.3 <fuelGauge>

The fuel gauge is a narrow, vertical gauge with a needle that moves from left to right to indicate a value within a specific range. The gauge distributes marks or "ticks" proportionally across its range. The meter attributes *rangeLower* and *rangeUpper* define the range, with the *rangeLower* value at left and the *rangeUpper* value at right.

The fuel gauge displays its current value in a text box at the center of the gauge.

You may specify warning lights to appear at the upper left or right of the gauge, above the ticks. These lights change color as the needle approaches them. The light at right changes color when the meter value rises to or above *thresholdUpper*. The light at left changes color when the meter value falls to or below *thresholdLower*. There is always a *thresholdLower* warning light with a default setting of 0. You must specify a *thresholdUpper* value if you want a warning light to appear at the top of the range as well.

<fuelGauge> has the following attributes:

Attribute	Description
Meter attributes	<fuelGauge> has the same general-purpose attributes as any meter. For descriptions, see the section "Meter Attributes."

Attribute	Description
highLampColor	String containing a CSS color value. This is the color that the <fuelGauge> displays when its *value* exceeds *thresholdUpper*. The default is a predefined Zen color that produces a “glowing” effect through shading: `url(#glow-red)` Several glow colors are defined in the class %ZEN.SVGComponent.svgPage. “<svgFrame>” always references this class, or some subclass of it, through its *svgPage* attribute, so these colors are always available to any SVG component in the frame.
logo	Text label for the face of the meter (similar to "Zen" in the illustration above). Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See “Zen Attribute Data Types.”
lowLampColor	String containing a CSS color value. This is the color that the <fuelGauge> displays when its *value* exceeds *thresholdUpper*. The default is a predefined Zen color: `url(#glow-red)`

2.4.4 <indicatorLamp>

The indicator lamp is a rectangular bar that changes its fill pattern depending on the meter *value*. Essentially there are three possible states: above *thresholdUpper*, below *thresholdLower*, or between the two values.

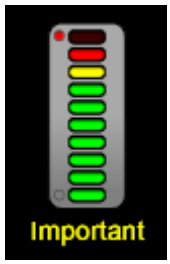
If the meter has a *label* attribute, this text appears in the middle of the indicator lamp with the style defined by *labelStyle*.

<indicatorLamp> has the following attributes:

Attribute	Description
Meter attributes	<indicatorLamp> has the same general-purpose attributes as any meter. For descriptions, see the section “Meter Attributes.”
highStyle	SVG CSS style definition. Specifies the fill style for the <indicatorLamp> when its *value* exceeds *thresholdUpper*. The *highStyle* must include a fill value, or mouse events within the shape do not work correctly. The default *highStyle* uses a predefined Zen color that produces a “glowing” effect through shading: `"fill: url(#glow-green);"` Several glow colors are defined in the class %ZEN.SVGComponent.svgPage. “<svgFrame>” always references this class, or some subclass of it, through its *svgPage* attribute, so these colors are always available to any SVG component in the frame.

Attribute	Description
lowStyle	SVG CSS style definition. Specifies the fill style for the <indicatorLamp> when its *value* falls below *thresholdLower*. The *lowStyle* must include a fill value, or mouse events within the shape do not work correctly. The default *lowStyle* uses a predefined Zen color: `"fill: url(#glow-red);"`
normalStyle	SVG CSS style definition. Specifies the fill style for the <indicatorLamp> when its *value* is between *thresholdLower* and *thresholdUpper*. The *normalStyle* must include a fill value, or mouse events within the shape do not work correctly. The default *normalStyle* uses a predefined Zen color: `"fill: url(#glow-blue);"`

2.4.5 <lightBar>



The light bar provides a stack of lamps arranged in a vertical bar. The light bar is similar to the traffic light, but its larger number of lamps provide a sense of movement from one end of the scale to the other.

The light bar is intended to appear “off” when the value is low, and fully lit when the value is high. The color of the bar shades from green (at the bottom of the scale) to yellow and red (at the top of the scale), implying that you should stop and address a problem when the value is high.

Sometimes a low number indicates a condition that requires attention. If this is the case you can reverse the sense of a light bar by inverting the values for *rangeLower* and *rangeUpper*. Then the fully lit lamp indicates low values, and the “off” lamp indicates high values.

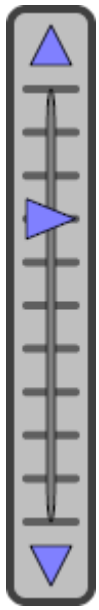
Warning lights appear at the top or bottom left of the light bar. The top light changes color when the meter value rises to or above *thresholdUpper*. The bottom light changes color when the meter value falls to or below *thresholdLower*. You can specify what color you want to use for these lights. You cannot change the colors for the light bar itself (green, yellow, red).

Note: When you reverse *rangeLower* and *rangeUpper*, do not at the same time reverse *thresholdUpper* and *thresholdLower*.

<lightBar> has the following attributes:

Attribute	Description
Meter attributes	<lightBar> has the same general-purpose attributes as any meter. For descriptions, see the section "Meter Attributes."
highLampColor	String containing a CSS color value. This is the color that the <lightBar> displays when its *value* exceeds *thresholdUpper*. The default is a predefined Zen color that produces a "glowing" effect through shading: `url(#glow-red)` Several glow colors are defined in the class %ZEN.SVGComponent.svgPage. "<svgFrame>" always references this class, or some subclass of it, through its *svgPage* attribute, so these colors are always available to any SVG component in the frame.
lowLampColor	String containing a CSS color value. This is the color that the <lightBar> displays when its *value* falls below *thresholdLower*. The default is a predefined Zen color: `url(#glow-red)`

2.4.6 <slider>



The slider is a vertical meter with a needle indicating a value between the *rangeLower* value at the bottom and the *rangeUpper* value at the top. The user can interact with the meter to edit its *value*, either by sliding the needle up and down with the mouse, or by clicking on the arrows at the top and bottom of the slider to increment the *value* along the tick marks.

<slider> has the following attributes:

Attribute	Description
Meter attributes	<slider> has the same general-purpose attributes as any meter. For descriptions, see the section "Meter Attributes."
constrained	If true, the slider *value* is constrained (rounded) to the nearest tick mark displayed within the slider. If false, the slider *value* is based on the exact position of the needle relative to *rangeLower* and *rangeUpper*. The default is true. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
tickMarks	Number of tick marks to display between the *rangeLower* and *rangeUpper*. The minimum is 0. The default is 10.

2.4.7 <smiley>

The smiley is the familiar yellow circle with two eyes and a smile. The mouth line changes depending on the meter *value*. The mouth forms a smile when the meter value is near *rangeUpper*. The mouth is a horizontal line at the midpoint between *rangeLower* and *rangeUpper*. The mouth forms a frown when the meter value is near *rangeLower*. This is useful when a large number indicates a positive condition, and a small number indicates a negative condition.

You can reverse the sense of a smiley by inverting the values for *rangeLower* and *rangeUpper*. Then the smile occurs at low values, and the frown at high values.

<smiley> has the same general-purpose attributes as any meter. For descriptions, see the section "Meter Attributes." <smiley> ignores any threshold values.

2.4.8 <speedometer>

The speedometer is a circular gauge with a needle that moves from lower left, all the way around to lower right, to indicate a value within a specific range. The gauge distributes marks or "ticks" proportionally across its range. The meter attributes *rangeLower* and *rangeUpper* define the range, with the *rangeLower* value at left and the *rangeUpper* value at right.

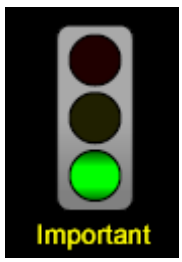
By default, the speedometer displays its current value in a text box at the center of the gauge. However, you can specify that this text box actually contains a different value that you control separately. Use the *independentOdometer* and *odometerValue* attributes for this.

You may specify warning lights to appear at the lower left or lower right of the gauge. These lights change color as the needle approaches them. The light at right changes color when the meter value rises to or above *thresholdUpper*. The light at left changes color when the meter value falls to or below *thresholdLower*.

<speedometer> has the following attributes:

Attribute	Description
Meter attributes	<speedometer> has the same general-purpose attributes as any meter. For descriptions, see the section "Meter Attributes."
highLampColor	String containing a CSS color value. This is the color that the <speedometer> displays when its *value* exceeds *thresholdUpper*. The default is a predefined Zen color that produces a "glowing" effect through shading: `url(#glow-red)` Several glow colors are defined in the class %ZEN.SVGComponent.svgPage. "<svgFrame>" always references this class, or some subclass of it, through its *svgPage* attribute, so these colors are always available to any SVG component in the frame.
independentOdometer	If true, this meter can display an additional value independent of its needle value, in a text box at the center of the gauge. If false, the value in the text box is the same as the needle value. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
logo	Text label for the face of the meter (similar to "Zen" in the illustration above). Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
lowLampColor	String containing a CSS color value. This is the color that the <speedometer> displays when its *value* falls below *thresholdLower*. The default is a predefined Zen color: `url(#glow-red)`
odometerValue	If *independentOdometer* is true, this is the value to display in the text box at the center of the gauge.

2.4.9 <trafficLight>



The traffic light consists of three circular lamps in a vertical column. From top to bottom, the lamps are red, yellow, and green.

When the meter value is at or below *thresholdLower*, the bottom lamp shows green. At values above *thresholdLower* but below *thresholdUpper*, the center lamp shows yellow. At values at or above *thresholdUpper*, the top lamp shows red. This is useful when a large number indicates a condition that requires attention.

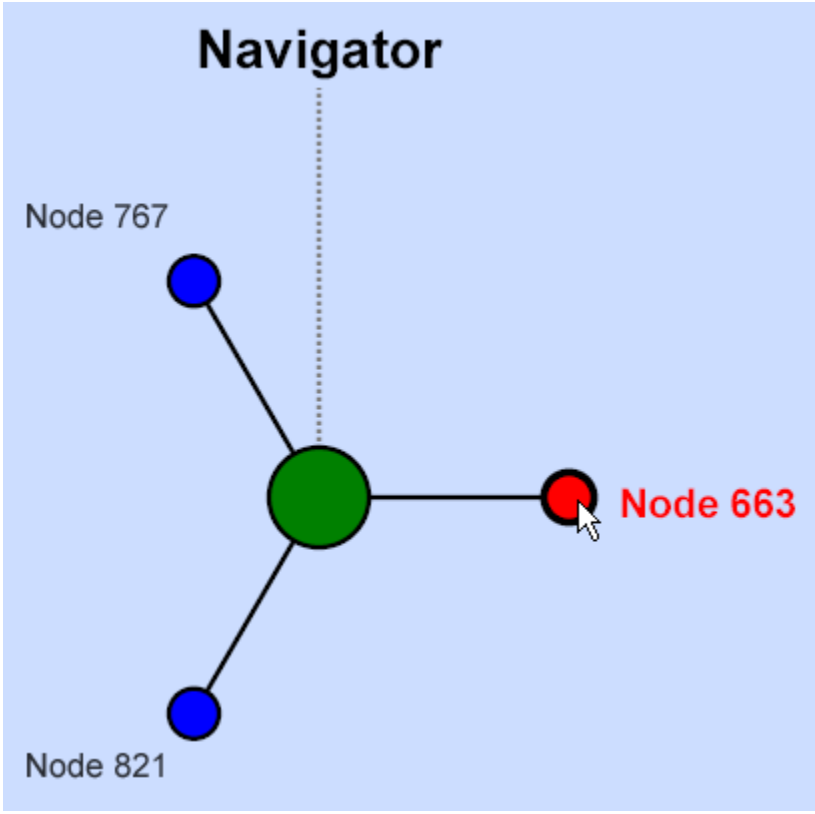
Sometimes a low number indicates a condition that requires attention. If this is the case you can reverse the sense of a traffic light by inverting the values for *rangeLower* and *rangeUpper*. Then the red lamp indicates low values, and the green lamp indicates high values.

<trafficLight> has the same general-purpose attributes as any meter. For descriptions, see the section "Meter Attributes." <trafficLight> looks best when *width* is half of *height*. You cannot change the colors for the traffic light.

2.5 Charts

Charts are SVG components that represent a series of data points. Zen provides several built-in chart types, including line charts, bar charts, and pie charts. As SVG components, charts have the SVG layout and style characteristics described in this chapter. However, charts also have many unique attributes. For a complete explanation, see the chapter "Zen Charts."

2.6 <radialNavigator>



The radial navigator is a specialized SVG component that displays the relationship between a set of data items as a dynamic, radial diagram. There is a central circular hub surrounded by a set of evenly spaced nodes. As you click on a node at the outer rim of the diagram, it becomes the hub node and the nodes to which it connects are shown circling it.

You define the nodes in the <radialNavigator> data set by providing <radialNode> elements inside the <radialNavigator> element. For example:

XML

```
<radialNavigator id="navigator" mainLabel="Navigator"
                 height="500" width="500" >
  <radialNode caption="Node 1" style="fill: green;"/>
  <radialNode caption="Node 2"/>
  <radialNode caption="Node 3"/>
</radialNavigator>
```

Attribute	Description
SVG component attributes	<radialNavigator> has the same general-purpose attributes as any SVG component. For descriptions, see the section "SVG Component Attributes."
backgroundStyle	SVG CSS style definition for the background panel. (Styles within SVG are CSS compliant, but there is a different set of styles available.)
hubStyle	SVG CSS style definition. Specifies the style for the central hub. This style must include a fill value, or mouse events within this shape do not work correctly.
mainLabel	Label text for the central hub. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
mainLabelStyle	SVG CSS style definition. Specifies the style for the *mainLabel* text.
nodeStyle	SVG CSS style definition. Specifies the style for the radial nodes. This style must include a fill value, or mouse events within this shape do not work correctly.
onselectNode	The *onselectNode* event handler for the <radialNavigator>. Zen invokes this handler whenever the user clicks the mouse on a node shape. See "Zen Component Event Handlers."
selectedIndex	When the *onselectNode* event is invoked, the *selectedIndex* contains the index (0-based) of the currently selected node. If the user clicks on the central hub node, *selectedIndex* is -1. When *selectedIndex* is –2, this means no node is selected.
title	Title text for the radial navigator. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
titleStyle	SVG CSS style definition. Specifies the style for the *title* text.

The <radialNode> element is the XML projection of the %ZEN.Auxiliary.radialNode class. <radialNode> has the following attributes:

Attribute	Description
caption	Text specifying the caption to display for this radial node. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
onclick	The *onclick* event handler for the <radialNode>. Zen invokes this handler whenever the user clicks the mouse on the node shape. See "Zen Component Event Handlers."
style	SVG CSS style definition for the node. The *style* must include a fill value, or mouse events within the node shape do not work correctly.
value	Value associated with this node. The value may be a decimal or integer number, but it is represented as a string.

When you work with %ZEN.SVGComponent.radialNavigator programmatically, you work with <radialNode> elements as members of the radialNavigator nodes property, a list collection of %ZEN.Auxiliary.radialNode objects. Each <radialNode> in the <radialNavigator> becomes a member of the nodes collection in radialNavigator, associated with its ordinal position: 1, 2, 3, etc.

2.7 <ownerDraw>

<ownerDraw> is an empty SVG component whose contents are filled dynamically by invoking a runtime callback method that provides SVG content. <ownerDraw> has the following attributes:

Attribute	Description
SVG component attributes	<ownerDraw> has the same general-purpose attributes as any SVG component. For descriptions, see the section "SVG Component Attributes."
onrender	The *onrender* event handler for the <ownerDraw>. See "Zen Component Event Handlers." This method provides the statements that render the SVG component on the page.

The following is an example of an <ownerDraw> element:

XML

```
<ownerDraw id="owner1"
           height="200" width="400"
           onrender="zenPage.doOwnerDraw(zenThis);"/>
```

In the above example, the JavaScript expression invokes the page class **doOwnerDraw** method with the <ownerDraw> object as its argument. The expression represents the page with the built-in variable `zenPage`, and the <ownerDraw> object with the built-in variable `zenThis`. Elsewhere in the page class, the **doOwnerDraw** method could look like this:

Class Member

```
ClientMethod doOwnerDraw(svg) [ Language = javascript ]
{
  // clear contents of ownerDraw component
  svg.unrender();

  // create a line; add it to the svg component
  for (var n = 0; n < 30; n++) {
    var line = svg.document.createElementNS(SVGNS,'line');
    line.setAttribute('x1',200);
    line.setAttribute('y1',100);
    line.setAttribute('x2',Math.random() * 400);
    line.setAttribute('y2',Math.random() * 200);
    line.setAttribute('style','stroke: blue; stroke-width: 2;');
    svg.svgGroup.appendChild(line);
  }
}
```

The SAMPLES namespace offers <ownerDraw> examples in the page classes ZENDemo.SVGBrowser and ZENTest.SVGOwnerDrawTest.

You can also create custom SVG components. See the "Custom Components" chapter in *Developing Zen Applications*

3

Zen Charts

This chapter explains how to place a chart on a Zen page. Every chart is an SVG component as described in the chapter "Zen and SVG" and is derived from the class %ZEN.SVGComponent.svgComponent. Charts follow the layout and style conventions for SVG components, but add specific behaviors of their own.

Zen reports callback charts use the same syntax as Zen pages to display charts. Callback charts are defined in the package %ZEN.Report.Display.COSChart, and all of the chart element names begin with the letter "c", for example, <cbarChart>. See "Using Callback Charts in Zen Reports". Syntax described in sections "Types of Chart," "Chart Layout, Style, and Behavior" and "Chart Axes" applies equally well to charts in Zen pages and callback charts in Zen reports. XPath charts is an older implementation of charting in Zen reports which uses syntax different from Zen pages. See "Using XPath Charts in Zen Reports".

Techniques for providing data for charts in Zen pages are different from both callback charts and XPath charts in Zen reports. See "Providing Data for Zen Page Charts" in this book and "Providing Data for Zen Report Charts" in the book *Using Zen Reports*.

The base chart class %ZEN.SVGComponent.chart defines the data, grid (axes and scales), styles, and legend used by charts. You can think of charts as occupying a virtual coordinate space that measures 100 units by 100 units. Internally, charts are plotted in terms of pixels, which allows you to specify size and positioning parameters in pixels as well as in terms of the virtual coordinate space. See "Specifying Size and Position." Within the total space occupied by the chart, there is a smaller plot area where the chart plots the data. Margins define the space around the plot area. Generally you use these margins as space in which to display the labels and legend for the chart.

This chapter begins by introducing the various types of chart, so that you can look at some visual examples and think about the items you want to display on the Zen page, before reading the exact details of how to use a specific type of chart. Chapter topics appear in this order:

- "Types of Chart"
- "Providing Data for Zen Page Charts"
- "Chart Layout, Style, and Behavior"
- "Chart Axes"

3.1 Types of Chart

This topic describes and illustrates the types of chart that you can place on the Zen page. Each description includes the unique attributes that define that type of chart. For attributes that all charts share in common, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."

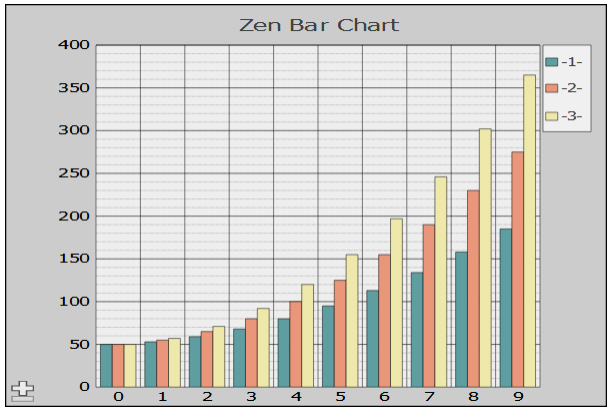
Zen offers the following built-in chart types:

- "Bar Charts"
- "Bubble Charts"
- "Bullseye Charts"
- "Combo Charts"
- "Difference Charts"
- "High/Low Charts"
- "Line Charts"
- "Percent Bar Charts"
- "Pie Charts"
- "Scatter Charts"
- "Tree Map Charts"

3.1.1 Bar Charts

A <barChart> displays one or more data series as a set of vertical or horizontal bars. The following figure shows a Zen bar chart displaying data from three data series.

Figure 3–1: Bar Chart



<barChart> has the following attributes:

Attribute	Description
Chart attributes	<barChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior." The *plotToEdge* attribute is always false for <barChart>. The <yAxis> *baseValue* attribute is used to plot the bottom of the bars. For details, see the section "Chart Axes."
chartStacked	If true, the bars at each position along the category axis are stacked atop one another (values are additive). See "chartStacked."
endTime	The end time for a *timeBased* x-axis.
showMultiples	If true, display the chart as a number of small charts, one per data series. See "showMultiples".
startTime	The start time for a *timeBased* x-axis.
timeBased	Specifies that the x-axis is a time line. See "timeBased."

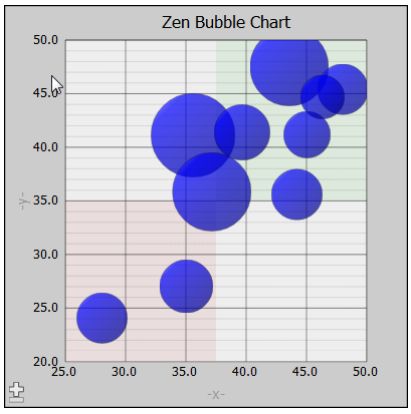
3.1.2 Bubble Charts

A <bubbleChart> displays data as circles or "bubbles" positioned at x and y coordinates. It requires a minimum of two data series. The first series supplies x values and the second supplies y values. When used to chart only two data series, <bubbleChart> is effectively an <xyChart>. <bubbleChart> does not support *chartPivot*

If you supply a third series, the chart uses that data to draw the radius of each bubble. Values in the radius series are scaled and multiplied by the value of the *radius* property.

The following figure shows a chart with x, y and radius data series. Note how the default partial transparency of the bubbles lets you see the shapes even when they overlap. The property *opacity* controls this characteristic. Note also that in this example, the value of *showQuadrant* is true, so the chart area is divided into quadrants.

Figure 3–2: Radius Data Series



You can supply up to two additional data series. The fourth series determines how colors are applied to the data bubbles, and the fifth controls opacity of bubbles.

Values in the fourth, or color data series can be any arbitrary data. The chart establishes color categories for each new value encountered in the series, and assigns the colors to corresponding bubbles on the chart. The chart uses the colors that are currently in effect, see *seriesColors* and *seriesColorScheme*.

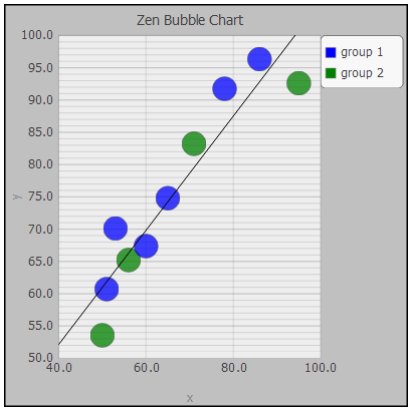
The following code fragment shows the part of an *ongetData* handler that sets up a color data series.

Class Member

```
/// Callback to provide data for bubble chart.
ClientMethod getBubbleChartData(series) [ Language = javascript ]
{
  var chart = zenPage.getComponentById('chart');
  var data = new Array(chart.seriesSize);

  // ...code omitted...

  if (series == 3) // color
  {
    for (var i = 0; i < chart.seriesSize; i++)
    {
      data[i] = (i%3)?"group 1":"group 2"
    }
  }
  return data;
}
```

The following figure shows the resulting chart, with colors applied to randomly generated data. Note that in this chart, the value of *showRegression* is true, so the chart contains a computed linear regression line, and *showQuadrant* is false, so the chart does not show quadrants. The property *lineStyle* provides a style specification for the regression line.

Figure 3–3: Color Data Series



Values in the fifth, or opacity data series can be any arbitrary data. Values in the opacity series are scaled and multiplied by the value of the *opacity* property. Values are not normalized, so a large range in values can result in the smallest values being driven to 0 and becoming invisible.

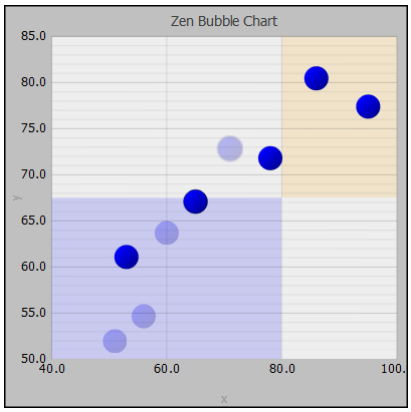
Class Member

```
/// Callback to provide data for bubble chart.
ClientMethod getBubbleChartData(series) [ Language = javascript ]
{
  var chart = zenPage.getComponentById('chart');
  var data = new Array(chart.seriesSize);

  // ...code omitted...

  if (series == 4) // opacity
   {
     data[1] = 1;
     data[2] = 4;
     data[3] = 1;
     data[4] = 1;
     data[5] = 4;
     data[6] = 1;
     data[7] = 4;
```

```
      data[8] = 4;
      data[9] = 4;
   }
   return data;
}
```

The following figure shows the resulting chart, with opacity applied to random generated data. Note also that the quadrants have been modified by setting the property *xCenterValue*, and the properties *upperRightStyle* and *lowerLeftStyle* have been used to modify the quadrant colors.

Figure 3–4: Opacity Data Series



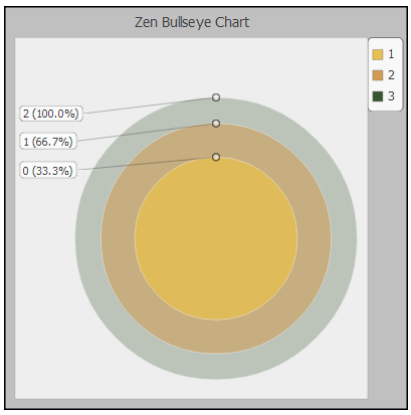
<bubbleChart> has the following attributes:

Attribute	Description
Chart attributes	<bubbleChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."
lineStyle	SVG CSS style definition applied to the regression line.
lowerLeftStyle	CSS style value to be applied to the lower left quadrant background.
opacity	Default opacity (from 0 to 1) for bubbles. If you have provided an opacity series, its values are scaled and multiplied by this value. Values in the opacity series are not normalized.
radius	Default radius (in logical units) for bubbles. If you have provided a radius series, its values are scaled and multiplied by this value.
showQuadrant	If true, then divide the plot area into quadrants, and apply color to the upper right and lower left quadrants. The default value is true, and the default colors are red in the upper right quadrant, and green in the lower left..
showRegression	If true, then draw a computed linear regression line.
xCenterValue	The x-axis value to use when drawing background quadrants. The default value is the center of the x-axis.
yCenterValue	The y-axis value to use when drawing background quadrants. The default value is the center of the y-axis.
upperRightStyle	CSS style value to be applied to the upper right quadrant background.

3.1.3 Bullseye Charts

A <bullseyeChart> displays data items as concentric circles.

Figure 3–5: Bullseye Chart



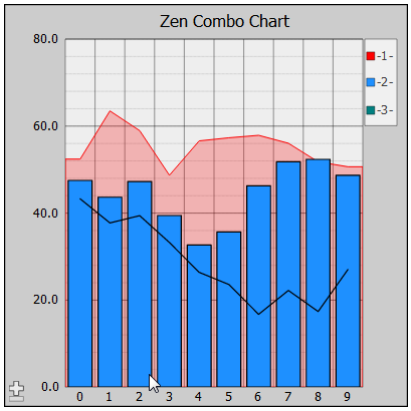
Bullseye charts are often used in situations where the largest circle shows the entire data set, the next circle shows data items that meet some relevant set of criteria, and the next circle shows data items that meet additional criteria. For example,

if the largest circle shows the total number of patients in a study, the next circle might show patients with diabetes, and the smallest diabetic patients undergoing a specific treatment.

3.1.4 Combo Charts

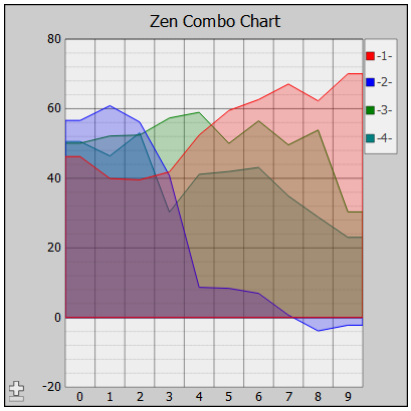
A <comboChart> displays multiple data series in a single chart, using area, bar and line charts. The following figure illustrates all three chart types.

Figure 3–6: Combo Chart Displaying Area, Bar and Line Charts



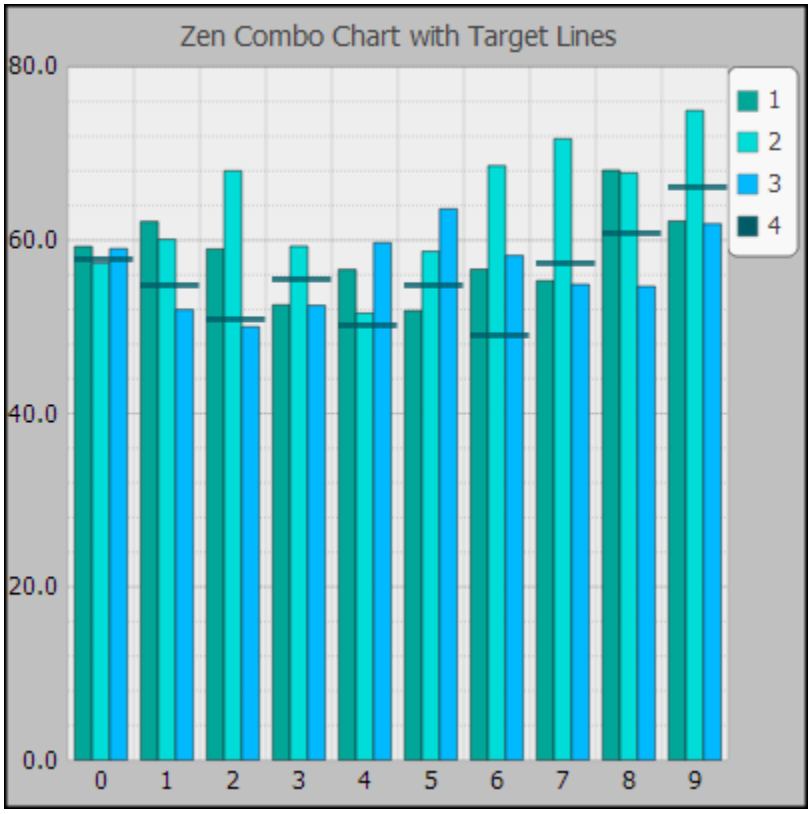
You can use other chart elements to plot multiple data series using a single type of chart, but <comboChart> sometimes offers advantages. For example, you can chart multiple filled line charts using <lineChart>, but <comboChart> may be a better choice, because it renders area charts with partial transparency, so all of the chart areas are visible, even when they overlap.

Figure 3–7: Combo Chart Displaying Four Area Charts



You can also use the attribute *seriesTypes* to display one of the data series as target lines. The following figure shows a chart where the fourth data series supplies target lines.

Figure 3–8: Combo Chart with Target Lines



<comboChart> supports the following attributes:

Attribute	Description
Chart attributes	<comboChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."
chartPivot	If true, pivot the chart (display categories vertically and values horizontally). See "chartPivot."
endTime	The end time for a *timeBased* x-axis.
lineStyle	SVG CSS style definition applied to lines in the chart.
seriesTypes	Comma-delimited list of types that indicates how the corresponding series in the chart should be displayed. Possible types are: "line", "area", "bar", and "target". The default type is bar. If the type is "target", the corresponding data series is used to put target lines on the chart. Area charts are rendered with partial transparency and placed behind all other chart types. Line charts are rendered in front of bar charts.
startTime	The start time for a *timeBased* x-axis.
timeBased	Specifies that the x-axis is a time line. See "timeBased."

3.1.5 Difference Charts

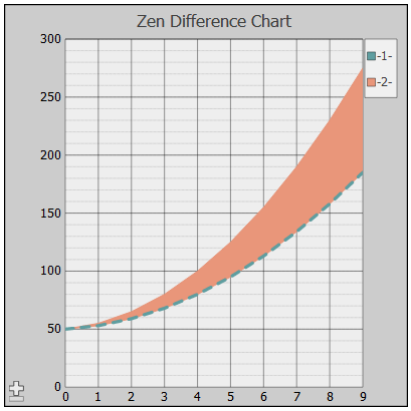
A <diffChart> is a specialized type of line chart that highlights the difference between two data series:

1. The first series provides a set of reference data values.
2. The second data series provides values that you want to compare to the reference data set.

The <diffChart> shades the area between the two series using the color of the second data series. To further distinguish between the series, the chart draws a line representing the reference series across the shaded area of the chart. This line uses the color of the first, or reference data series, and can take additional styling from the *refLineStyle* attribute. <diffChart> does not support *chartPivot*.

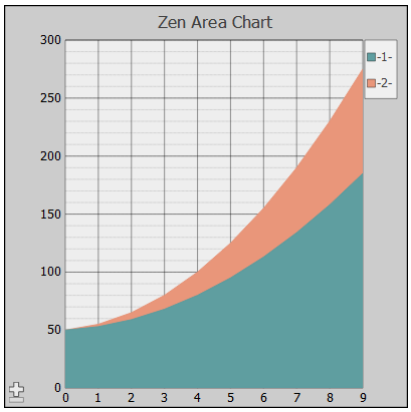
The following figure shows a difference chart:

Figure 3–9: Difference Chart



For comparison, the following figure shows the same data plotted as a filled line (area) chart:

Figure 3–10: Area Chart



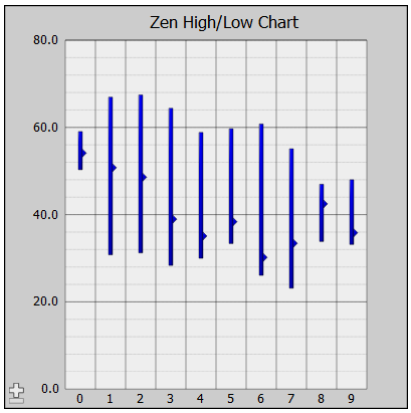
<diffChart> has the following attributes:

Attribute	Description
Chart attributes	<diffChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."

refLineStyle	SVG CSS style definition for the reference line. By default, this line uses the color assigned to the reference (first) data series, but typically it also assigns some patterning using *refLineStyle*. The default is:
	`stroke-dasharray: 1,1;`
	The difference chart in the previous figure shows:
	`stroke-dasharray: 2,2; stroke-width: 1%;`

3.1.6 High/Low Charts

Figure 3–11: High/Low Chart



A <hilowChart> can be used to show stock market high-low-close values, or to trace a measured value along with its high and low ranges to indicate possible error margins. The chart displays a set of bars as established by three data series:

1. A series of high values sets the top (right) limit of each bar.
2. A series of low values sets the bottom (left) limit of each bar.
3. (Optional) The "closing" values. The chart places a marker on each bar at these values.

Each low value is assumed to be smaller than its corresponding high value. Each closing value is assumed to be between its corresponding high and low values. The chart uses its first *seriesColors* value to plot all bars and marker. It ignores the colors provided for the other series.

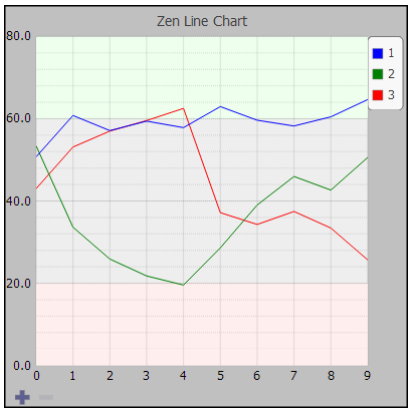
<hilowChart> has the following attributes:

Attribute	Description
Chart attributes	<hilowChart> has the same general-purpose attributes as any Zen chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior." The *plotToEdge* attribute is always false for <hilowChart>.
chartPivot	If true, pivot the chart (display categories vertically and values horizontally). See "chartPivot."
invertedBarStyle	SVG CSS style definition. Specifies the style used for bars where the high value is less than the low value.

3.1.7 Line Charts

A <lineChart> displays one or more data series as a set of lines. The following figure shows a simple line chart.

Figure 3–12: Line Chart



<lineChart> has the following attributes:

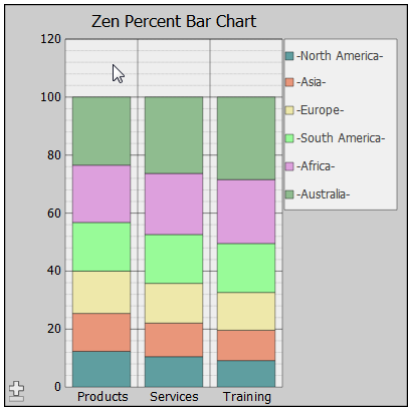
Attribute	Description
Chart attributes	<lineChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."
chartFilled	If true, the area under each line is filled (as in an area chart). If false, it is not filled (as in a line chart). This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
chartPivot	If true, pivot the chart (display categories vertically and values horizontally). See "chartPivot."
chartStacked	If true, the bars at each position along the category axis are stacked atop one another (values are additive). See "chartStacked."
endTime	The end time for a *timeBased* x-axis.
lineStyle	SVG CSS style definition applied to lines in the chart. A color specified in this attribute takes precedence over colors specified in *seriesColors*.
showMultiples	If true, display the chart as a number of small charts, one per data series. See "showMultiples."
startTime	The start time for a *timeBased* x-axis.
timeBased	Specifies that the x-axis is a time line. See "timeBased."

3.1.8 Percent Bar Charts

A <percentbarChart> displays each data series as a bar in the chart. All the bars are the same height, and represent 100% of the values in the series. Bands in the bar represent items in the series, and are sized proportional to that items contribution to the total. This method of handling data series is similar to a <pieChart> when `plotBy="both"`, except that the pie chart also shows the relative contribution of each series. See "Pie Charts by Items."

You can use *ongetLabelX* to provide labels for the bars, and *seriesNames* to provide labels for the legend, matching names to colors by order in the series.

Figure 3–13: Percent Bar Chart



<percentbarChart> has the following attributes:

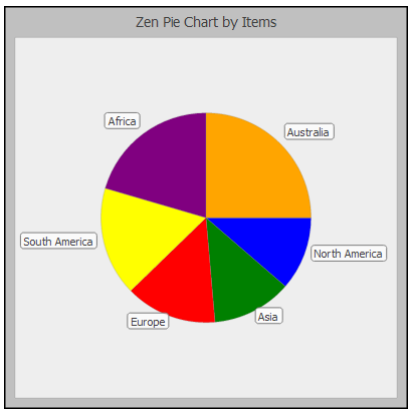
Attribute	Description
Chart attributes	<percentbarChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."
chartPivot	If true, pivot the chart (display categories vertically and values horizontally). See "chartPivot."

3.1.9 Pie Charts

Pie charts can plot single or multiple data series, and can show the series, the items in the series, or both. <pieChart> does not support axes or grids in the plot area, as do line charts or bar charts.

A <pieChart> that plots the items in a single data series displays a circle with radial slices representing items in the series. The chart adjusts the size of each slice to be proportional to the contribution of that item to the total. Note that the end user can rotate the pie chart with click and drag gestures in the browser.

Figure 3–14: Pie Chart with One Data Series



<pieChart> has the following attributes:

Attribute	Description
Chart attributes	<pieChart> has the same general-purpose attributes as any Zen chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."
holeSize	Controls whether a hole is displayed in the center of the pie chart, and if so, how big the hole is. The value of this property is a percentage of the chart's radius along the x-axis. If the size of the chart changes, the size of the center hole maintains its proportional size. The value can range from 0 to 0.9. Values larger than 0.9 have no additional effect. The default value is 0.
pieHeight	Controls the apparent height of 3D pie charts. The value of this property is a percentage of the chart's radius along the x-axis. If the size of the chart changes, its depth maintains its proportional size. The value can range from 0 to 1. The default value is 0.33.
pieScale	Decimal value that specifies the scaling factor for the size of the pie within the chart. The default scaling value is 1.0. A value larger than 1.0 makes the pie bigger relative to the plot area; a value smaller than 1.0 makes the pie smaller.
plotBy	Specifies how the pie chart plots its data. Possible values are "items", "series", "both", or "auto". The default is "auto". See "plotBy."
rotateBy	If specified, rotate the pie chart by this amount (in degrees).
showMultiples	If true, display the chart as a number of small charts, one per data series. See "showMultiples."
showPercentage	If true, percentage values (rounded to nearest integer) are displayed as part of the label for each slice. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

3.1.9.1 plotBy

The <pieChart> *plotBy* attribute controls how Zen generates a pie chart from the data provided. You may actually provide multiple series to the <pieChart>. The chart processes all the data series to create one series, and then displays that series as slices in the chart. *plotBy* options are as follows.

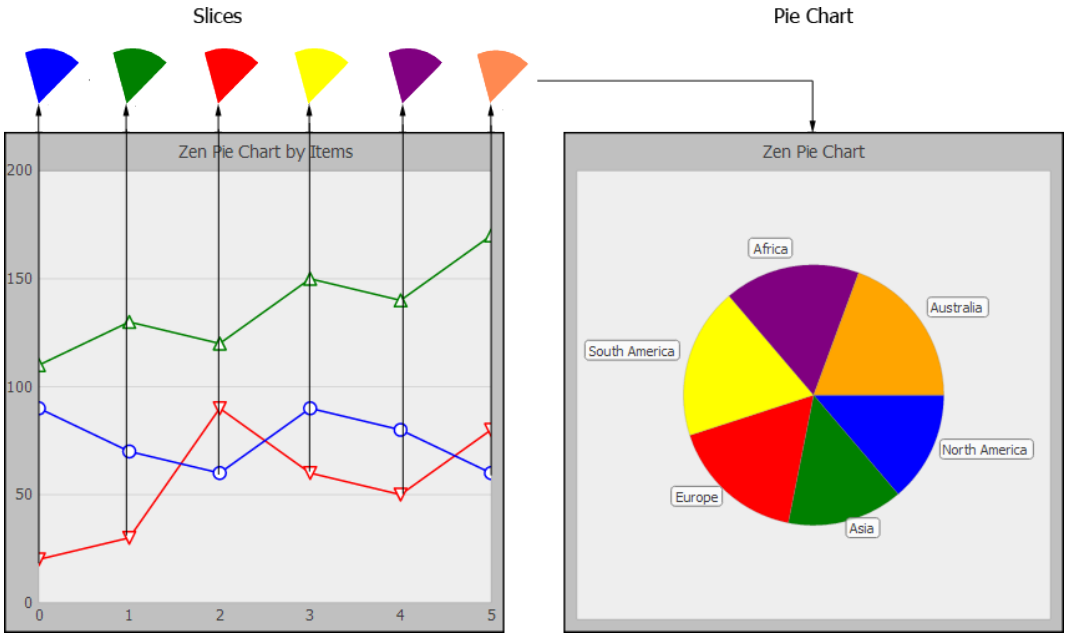
- "items" – plot a slice for every item in this chart's data.
- "series" – plot a slice for every data series in this chart's data.
- "both" – plot a slice for each item within each data series, which means that the chart contains seriesCount * seriesSize slices.
- "auto" – automatically select the appropriate way to display data based on how many data series and items are present.

Pie Charts by Items

When the value of *plotBy* is "items", a <pieChart> plots one slice for every item within the data series. If you provide multiple data series to an "items" pie chart, each slice of the pie represents the total of a particular item, summed across all of the data series in the chart.

The following conceptual figure shows how three data series, each containing six items, would generate an six-slice pie chart. Each slice represents the sum of the values for that item provided by the three series.

Figure 3–15: How Zen Plots Pie Charts by Item



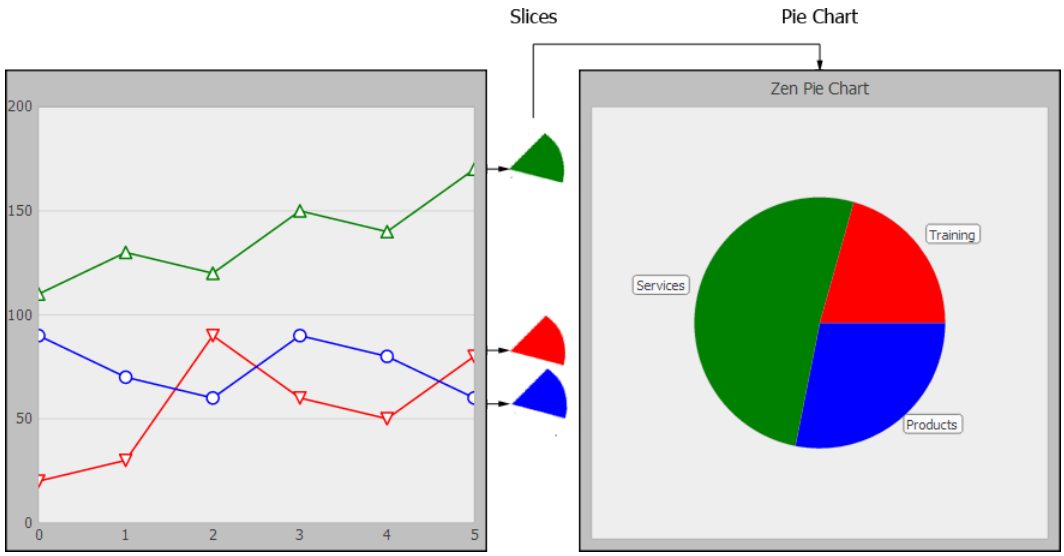
When the value of *plotBy* is "items", labels for the slices are treated as <yAxis> labels. This means you cannot specify them directly using a chart attribute such as *seriesNames*. Labels are provided by a %ZEN.Auxiliary.dataController, if present, or you can provide an *onGetLabelY* event handler to get the label values.

Pie Charts by Series

When the value of *plotBy* is "series", multiple series are in use. The <pieChart> plots one slice for every data series, so the number of slices in the pie chart is *seriesCount*. Each slice represents the sum of all the items within one of the series. The *seriesNames* attribute provides the labels for the slices, and for the legend.

The following conceptual figure shows how three data series, each containing six items, generate a three-slice pie chart. Each slice represents the sum of the eight items in that series.

Figure 3–16: How Zen Plots Pie Charts by Series

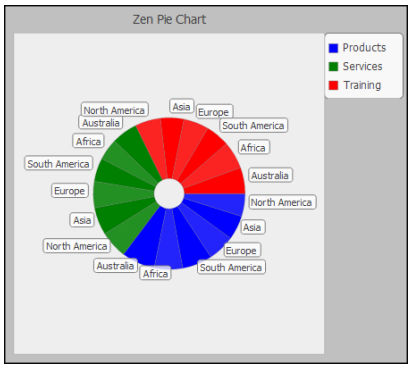


Pie Charts by Both Items and Series

When the value of *plotBy* is "both", multiple series are in use. The <pieChart> plots one slice for every item in every data series, so there are *seriesCount* times *seriesSize* slices. The base color for each slice is the associated series color. Alternating slices use dark and normal shades of this color. The chart legend displays series names; the *seriesNames* attribute provides these labels. The slices display item names; the labels are provided by an *onGetLabelY* event handler, or a data controller.

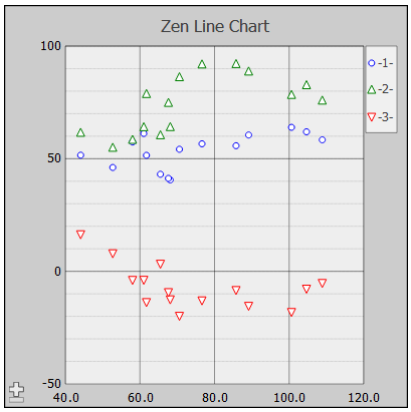
The following pie chart example compares three series (Products, Services, and Training), each of which has data items in four categories (America, Asia, Europe, and Africa). The chart has twelve slices.

Figure 3–17: Zen Pie Chart from Both Items and Series



3.1.10 Scatter Diagrams

Figure 3–18: XY or Scatter Chart



An <xyChart> plots two or more data series as (x,y) points. This type of chart is sometimes called a scatter diagram; it is intended to show the raw data distribution for the series. The <xyChart> represents its data series as follows:

1. The first data series provides the x values
2. The second data series provides correlated y values
3. Any additional data series are plotted as y values correlated to the x values provided by the first series.

The result is that an <xyChart> always displays one less plot than its number of data series.

<xyChart> has no unique attributes. <xyChart> has the same general-purpose attributes as any Zen chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior." An <xyChart> always has *markersVisible* and *plotToEdge* set to true. Generally you need to manipulate some of the other chart attributes to produce the desired results. For example:

- The *seriesCount* value must always be one more than the number of plots you want to display. This leaves room for the first, x-value series.
- The first data series is not plotted, so the <xyChart> applies series settings beginning with the second series, not the first. For example:
  - The *seriesColors* list applies to the second, third, and successive series.
  - The *seriesNames* list applies to the second, third, and successive series.
  - The *markerShapes* list applies to the second, third, and successive series.
- A scatter diagram does not appear "scattered" unless you hide the lines between the markers. To do this, set the *plotStyle* for an <xyChart> as follows:

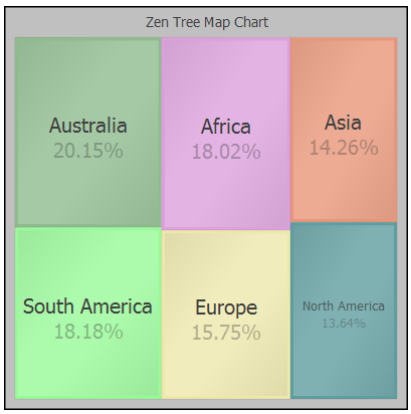
  `plotStyle="stroke:none;"`. This is a better approach than setting `plotStyle="stroke-width: 0;"`, because some SVG implementations render a very thin line, even when stroke-width is set to 0.

<xyChart> has the following attributes:

Attribute	Description
Chart attributes	<xyChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."
chartPivot	If true, pivot the chart (display categories vertically and values horizontally). See "chartPivot."
independentXSeries	If false, the first data series is used to supply x values for the chart and all other data series provide y values. If true, then the chart displays multiple x series. In this case, the first data series provides the first set of x values, the second data provides the first set of y values, the third data series provides the second set of x values, and so on. This property is optional, and the default value is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

3.1.11 Tree Map Charts

Figure 3–19: Tree Map Chart



A <treeMapChart> plots the items in a single data series and displays each item in the series as a rectangle and arranges them in a larger rectangle. The chart adjusts the size of each smaller rectangle to be proportional to the contribution of that item to the total. You can display more than one data series by setting the attribute *showMultiples* to true. Each of the multiple charts displays a different data series.

Attribute	Description
Chart attributes	<xyChart> has the same general-purpose attributes as any chart. For descriptions, see the sections "Providing Data for Zen Page Charts" and "Chart Layout, Style, and Behavior."
showMultiples	If true, display the chart as a number of small charts, one per data series. See "showMultiples."

3.2 Providing Data for Zen Page Charts

The data for a chart consists of one or more data series. Each data series that you use for a chart is a simple array of numeric values. You can provide the data for a Zen page chart in one of two ways:

- "Using a JavaScript Method" in the page class, by setting the chart's *ongetData* attribute.
- "Using a Data Controller", by setting the chart's *controllerId* attribute.

Regardless of which technique your chart uses to retrieve data, you can limit the data returned by either technique to the desired number and size of series.

Zen reports callback charts and Zen pages use identical syntax to display charts. Callback charts are defined in the package %ZEN.Report.Display.COSChart

When it comes to *displaying* charts, Zen pages and Zen reports use identical syntax. Syntax described in the previous section, "Types of Chart," and in the later sections "Chart Layout, Style, and Behavior" and "Chart Axes" applies equally well to Zen pages and Zen reports. However, the techniques for providing *data* for charts in Zen reports are different from the techniques for Zen pages. For details, see the section "Providing Data for a Zen Report Chart" in the book *Using Zen Reports*.

To provide data for charts on Zen pages, use the topics in this section.

3.2.1 Using a JavaScript Method

All charts support the *ongetData* attribute for generating chart data.

Attribute	Description
ongetData	Client-side JavaScript expression that Zen invokes whenever: • The chart is initially displayed • The chart's **updateChart** method is invoked For example: `ongetData="return zenPage.getChartData(series);"` In Zen reports, *ongetData* is a server-side method written in Object Script or other appropriate language.

The *ongetData* expression invokes a client-side JavaScript method defined in the page class. This method becomes the *ongetData* event handler for the chart. The chart calls the *ongetData* event handler once for each of the data series specified by *seriesCount*. The event handler accepts a single argument, *series*, which specifies the data series currently being processed. The value of *series* is the 0-based ordinal number of the current data series; possible values range from 0 to `seriesCount - 1`.

The event handler also needs to know how many values to supply for the data series. This value is provided by the *seriesSize* attribute of the chart. In the example code that follows this explanation, this line gets the value of *seriesSize* from the chart:

```
var data = new Array(chart.seriesSize);
```

The method puts `seriesSize - 1` items into an array and returns that array. It can use whatever approach is appropriate to provide data for the chart. For instance, it may use a switch statement based on the value of the *series* input argument.

The following is a sample *ongetData* event handler that provides random values as a test. You can see this event handler in the SAMPLES namespace in the template page class ZENTest.SVGChartTest. It provides the data for charts drawn by page classes that extend the ZENTest.SVGChartTest template, such as SVGLineChartTest and SVGBarChartTest:

Class Member

```
ClientMethod getChartData(series) [ Language = javascript ]
{
  var chart = zen('chart');
  var data = new Array(chart.seriesSize);
  var value = 50;

  for (var i = 0; i < chart.seriesSize; i++) {
    if (Math.random() > 0.9) {
      value += (Math.random() * 50) - 35;
    }
    else {
      value += (Math.random() * 20) - 9;
    }
    data[i] = value;
  }
  return data;
}
```

The previous example creates a chart with random data.

Normally, you want to create charts using real data from the server. In the typical case, your server-side method returns a list. On the client side, this list is consumed by the chart component that displays the data. One way to make this work is shown in the following example:

1. When our example page is first displayed, its **%OnAfterCreatePage** method calls a server-side method **PopulateData**() to place values into a page property called Population for later use by client-side code. In our example, this happens only once, when **%OnAfterCreatePage** is called. Later user actions may change how this data is viewed, but in this example, we only get data from the server once.

   Class Member

   ```
   Method %OnAfterCreatePage() As %Status
   {
     // Get the data while we're on the server
     Set sc = ..PopulateData()

     // Set the initial series size programmatically
     Set chart = %page.%GetComponentById("chartPop")
     Set chart.seriesSize = $Length(..States,",")
     Quit sc
   }
   ```

2. The Population property is defined in our page class as follows:

   Class Member

   ```
   Property Population As %ZEN.Datatype.list(DELIMITER = ",");
   ```

   And our server-side method **PopulateData**() places values and comma delimiters into the Population list as follows. Step 3 describes the Internals of the **GetCountByState**() class query referenced here:

   Class Member

   ```
   Method PopulateData() As %Status
   {
     Try {
       Set sc = $System.Status.OK()

       // Get a resultset containing population by state
       Set sc = ##class(SimpleZenChart.Person).GetCountByState(.rs)
       Quit:$System.Status.IsError(sc)

       // Populate the page properties with comma delimited values
       While (rs.%Next()) {
         Set ..Population = ..Population _ rs.%Get("PersonCount") _ ","
         Set ..States = ..States _ rs.%Get("HomeState") _ ","
       }

       // Remove trailing delimiter
       Set ..Population = $Extract(..Population,1,*-1)
       Set ..States = $Extract(..States,1,*-1)
     }
     Catch(ex) {
       Set sc = ex.AsStatus()
     }
     Quit sc
   }
   ```

3. As seen in the above excerpt, **PopulateData**() invokes a class query defined in SimpleZenChart.Person to retrieve the results that it uses in constructing the Population list. The **GetCountByState**() method looks like this:

   Class Member

   ```
   ClassMethod GetCountByState(Output Results As %SQL.StatementResult) As %Status
   {
     Try {
       Set sc = $System.Status.OK()

       // Make a new SQL statement
       // Use an array to create the query text (this is not required)
       // Pass in schema search list into statement constructor
   ```

```
    Set statement = ##class(%SQL.Statement).%New(,"SimpleZenChart")
    Set query = 4
    Set query(1) = "SELECT HomeState, COUNT(ID) As PersonCount"
    Set query(2) = "FROM Person"
    Set query(3) = "GROUP BY HomeState"
    Set query(4) = "ORDER BY HomeState"

    // Prepare query
    Set sc = statement.%Prepare(.query)
    Quit:$System.Status.IsError(sc)

    // Execute query
    Set Results = statement.%Execute()

    // Check %SQLCODE for an error
    If (Results.%SQLCODE < 0) {
      Set sc =
        $System.Status.Error($$$GeneralError,
        "Error in %Execute. %SQLCODE = "
        _Results.%SQLCODE_" Error message = "_Results.%Message)
      Quit
    }
  }
  Catch (ex) {
    Set sc = ex.AsStatus()
  }
  Quit sc
}
```

4. Each time a chart on our example page is refreshed, including the first time the page is displayed, the chart consults its *ongetData* expression to see which client-side method it should invoke to populate itself with data. In the following excerpt, the chart is a <barChart> and the client-side method is called **getData**():

   XML

```
<barChart
          id="chartPop" width="100%" height="100%"
          selectedItemStyle="fill:rgb(255,0,255);" seriesCount="1"
          appearance="2D"
          title="Population By State"
          ongetData="return zenPage.getData(series);"
          ongetLabelX="return zenPage.getXLabels(value);"
          onelementClick="zenPage.onSelectElement(zenThis);">
   <xAxis title="States"/>
 </barChart>
```

5. Our sample client-side method **getData**() consults the Population property of the client-side page object to get the list of values stored in that property. Because Population was defined as type %ZEN.Datatype.list in the page class, **getData**() automatically understands Population as a JavaScript array; no conversion is necessary. **getData**() returns this array as its return value.

   Class Member

```
ClientMethod getData(series) [ Language = javascript ]
{
  try {
    var data = zenPage.Population;
    return data;
  }
  catch (ex) {
    zenExceptionHandler(ex,arguments);
  }
}
```

6. The <barChart> updates its display using the values in the JavaScript array returned by **getData**().

7. This example offers other interesting features. For example, when you click on a bar in the bar chart, Zen fires the client-side method identified by the <barChart> *onelementClick* attribute. In our example, this client-side method is called **onSelectElement**(). It uses the JavaScript utility function zenSetProp() to find some of the other components on the page then change their contents using data acquired from client-side methods, including **getData**().

Class Member

```
ClientMethod onSelectElement(chart) [ Language = javascript ]
{
  try {
    var selected = chart.getProperty('selectedItem')

    // Set the population and count for the item
    zenSetProp('htmlState','content',this.getXLabels(selected));
    zenSetProp('htmlCount','content',(this.getData())[selected]);
  }
  catch (ex) {
    zenExceptionHandler(ex,arguments);
  }
}
```

8. In the excerpt above, the client-side method **onSelectElement**() uses the number of the currently selected bar in the bar chart as an index into the array returned by **getData**(). **getXLabels**() also uses an index into a JavaScript array, but in this case the array is the page property States, which contains a list of states whose population statistics are being stored in the example. Because the States property was defined as type %ZEN.Datatype.list in the page class, **getXLabels**() automatically understands States as a JavaScript array; no conversion is necessary.

Class Member

```
ClientMethod getXLabels(value) [ Language = javascript ]
{
  try {
    var data = zenPage.States
    return data[value];
  }
  catch (ex) {
    zenExceptionHandler(ex,arguments);
  }
}
```

9. A final item of note in this example is one that changes the series size for the chart without modifying or resending any data. The client-side method **changeSeriesSize**() is invoked when the user selects one of the choices in a list box on the same page with the bar chart. The list box is defined as follows:

XML

```
<listBox id="lbSeriesSize"
         label="Select Series Size"
         value="5"
         onchange="zenPage.changeSeriesSize(zenThis);">
  <option text="10"/>
  <option text="20"/>
  <option text="30"/>
  <option text="40"/>
  <option text="50"/>
</listBox>
```

10. The client-side method **changeSeriesSize**() uses the *text* value from the selected <option> and uses it to reset the *seriesSize* value for the <barChart> whose *id* is `chartPop`. This is our example <barChart> as shown in step 4. When the *seriesSize* is reset, the chart redisplays.

Class Member

```
ClientMethod changeSeriesSize(listbox) [ Language = javascript ]
{
  try {
    // Get selected index and compute value
    var selected = listbox.getProperty('selectedIndex');
    var value = listbox.getOptionText(selected);

    // Set the series size in the chart
    zenSetProp('chartPop','seriesSize',value);
  }
  catch (ex) {
    zenExceptionHandler(ex,arguments);
  }
}
```

3.2.2 Using a Data Controller

All charts support the following attributes, which associate the chart with a view on a data controller as described in the chapter "Model View Controller. "

Attribute	Description
controllerId	Identifies the data controller that provides the data for this chart. The *controllerId* value must match the *id* value provided for that <dataController>.
onnotifyView	The *onnotifyView* event handler for the <chart>. This attribute applies if the chart is associated with a data controller. Zen invokes this handler each time the data controller connected to this chart raises an event. See "Zen Component Event Handlers."

If your XData Contents block contains a <dataController> reference that looks like this:

XML

```
<dataController id="source" modelClass="myPackage.MyModel" modelId="1"/>
```

Then your XData Contents block may also contain a chart definition that looks like this:

XML

```
<pieChart id="myChart" controllerId="source"
          height="300" width="300"
          title="Pie Chart" titleStyle="fill: black;"
          backgroundStyle="fill: #c5d6d6;"
          plotAreaStyle="fill: white;"
          labelStyle="fill: black;"
          legendVisible="true" legendX="83" legendY="8"
          legendHeight="" legendWidth="15" >
</pieChart>
```

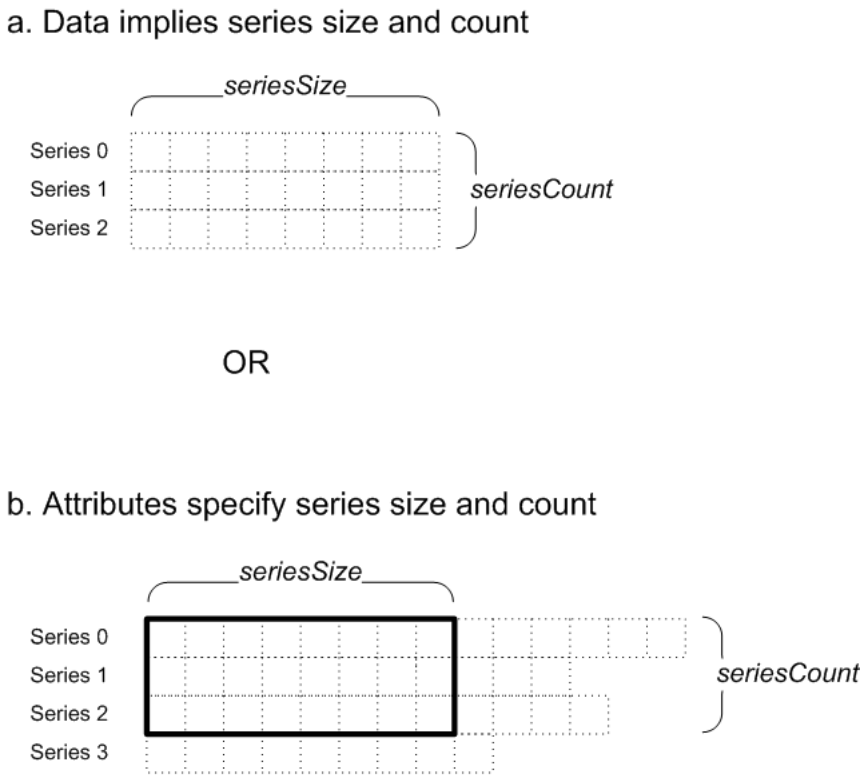
The chart's *controllerId* value must match the <dataController> *id* value. The chart takes its *seriesSize* from the number of properties in the <dataController> *modelClass*.

3.2.3 Limiting the Data Set

The following attributes from the base class %ZEN.SVGComponent.chart tell the chart how many series to use, and how many of the items in each data series to use, when constructing the chart. All types of chart support these attributes. If you do not use a *dataController* to provide the data set, you must specify both *seriesCount* and *seriesSize*. If you use a *dataController*, Zen determines the number of series and items from the data.

Attribute	Description
seriesCount	Positive integer specifying the number of series in the chart. If you do not use a *dataController* and your data source provides a larger number of series than you want to use, you can trim the number of displayed series by setting *seriesCount* to a smaller number. The following figures illustrate the action of *seriesCount*.
seriesSize	Positive integer specifying the number of items within each data series to display on this chart. If you do not use a *dataController* and your data source provides a larger number of items in the series than you want to use, you can trim the number of displayed items by setting *seriesSize* to a smaller number. The following figures illustrate the action of *seriesSize*.

Figure 3–20: Data Series Count and Size

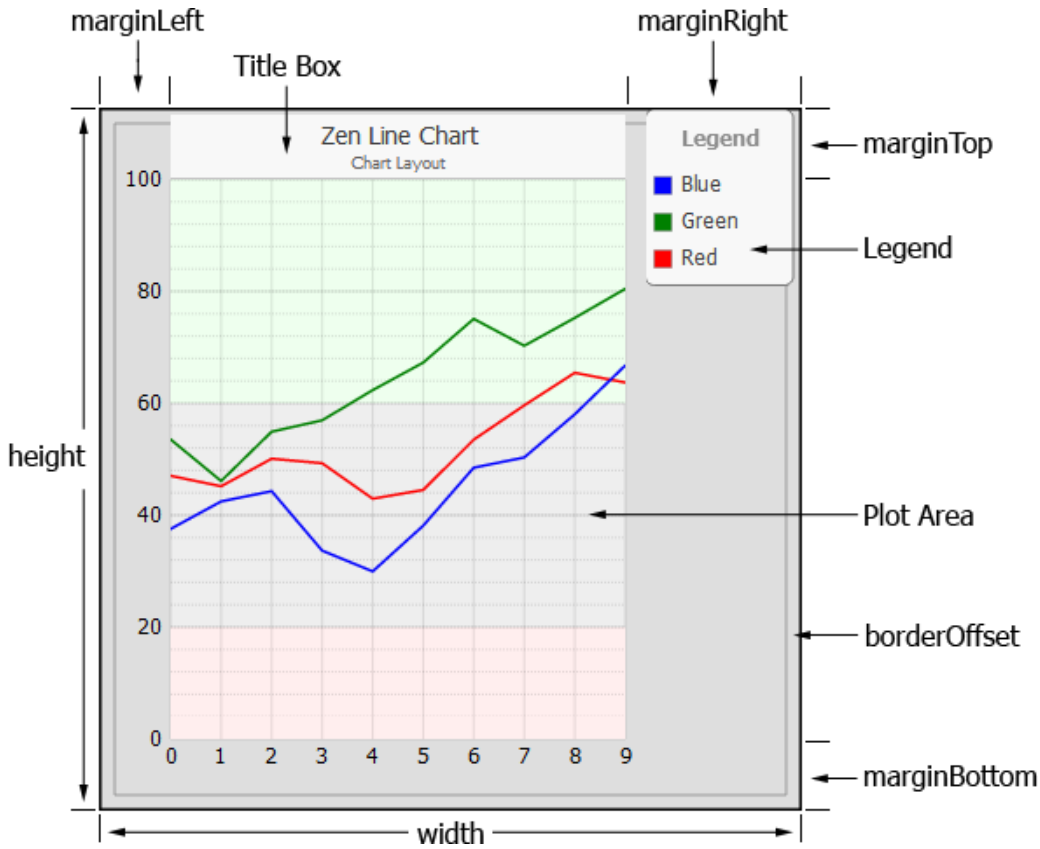


3.3 Chart Layout, Style, and Behavior

The following diagram shows the major components of a Zen chart. The chart properties described in the following sections control the positioning, style, and behavior of these components.

Chart properties include those from the SVG component class %ZEN.SVGComponent.svgComponent, such as *width* and*height*, and those from the base chart class %ZEN.SVGComponent.chart, such as *marginTop* and *borderOffset*.

Figure 3–21: Layout Attributes for Zen Charts



The base class %ZEN.SVGComponent.chart offers a large number of attributes that you can apply to a Zen chart to control details such as:

- "Layout and style" — The relative size and characteristics of the background
- "Plot area" — The part of the chart that displays the data
- "Markers" — Shapes that highlight the exact data points on a continuous plot
- "Legends" — A key to the meaning of each plot on the chart
- "Titles" — Text that labels the chart, and the items on the chart
- "User Selections" — How the chart should respond to user actions, such as mouse clicks
- "Chart Axes" — Characteristics of the two axes that define most charts

3.3.1 Specifying Size and Position

Unless the specific description of a property states otherwise, you can give values for properties that specify sizes and positions in one of the following ways:

- Do not specify a value, and let Zen calculate the value automatically.
- Use a value from 0 to 100 that is interpreted as a percentage of the current chart size.
- Specify a length value with units, such as "10px", to indicate that you want a margin of 10 pixels, independent of chart size

3.3.2 Layout and Style

The following attributes from the base class %ZEN.SVGComponent.chart determine the background style and the position of the plot area within the chart.

Table 3–1: Chart Layout and Style Attributes

Attribute	Description
SVG component attributes	A chart has the same attributes as any SVG component. For descriptions, see the "SVG Component Attributes" section in the chapter "Zen and SVG."
backgroundStyle	SVG CSS style definition. Specifies the style for the background panel. This is the area outside the plot area but inside the chart component's *height* and *width*.
borderOffset	SVG CSS style definition. Specifies the distance in pixels between the background rectangle and the border. The default value is 8.
borderRadius	Specifies the radius in pixels used to round the corners of the chart's background rectangle and border.
borderStyle	SVG CSS style definition. Specifies the style used for the border line which is inset from outer edge of chart.
hasZoom	If true, display zoom in/out buttons (for certain chart types). The default value is false. The property *scrollButtonStyle* controls the appearance of the buttons.
marginBottom	Margin to allow from the bottom edge of the background to the bottom edge of the plot area. Axis labels or the chart legend usually appear in this space. You can provide a value as specified in "Specifying Size and Position."
marginLeft	See *marginBottom* in this table.
marginRight	See *marginBottom* in this table.
marginTop	See *marginBottom* in this table.

3.3.3 Plot Area

The following attributes from the base class %ZEN.SVGComponent.chart determine display conventions for graphs within the plot area and for the coordinate axes that border the plot area. Also see the section "Chart Axes."

Attribute	Description
appearance	Controls the appearance of this chart. A value of "2D" provides standard two-dimensional chart appearance. A value of "3D" gives the chart a three-dimensional appearance so that the plot area appears to be recessed. Certain chart types, such as bar charts, display items with a 3D look. For pie charts, setting this property to "3D" displays a pie chart with a 3D appearance, but does not affect the plot area. The default value is "2D" for some charts and "3D" for others.

Attribute	Description
autoScaleText	Specifies how the chart handles text elements when it is resized. If true, the text scales in proportion to the chart and all requested labels are rendered regardless of legibility. If false, the size of the text elements is fixed with respect to the page and some labels may be omitted to avoid visual overlap if there is not enough space on the chart to render the all values. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
axisLineStyle	An optional SVG CSS style definition applied to line drawn for x- and y-axes.
axisTitleStyle	An optional SVG CSS style definition applied to the title specified by the *title* property of the axis.
bandLower	Decimal value. If defined, the chart displays a colored band on the plot area covering the range lower than this value. *bandLowerStyle* defines the style of this band.
bandLowerStyle	SVG CSS style definition for the band defined by *bandLower*.
bandUpper	Decimal value. If defined, the chart displays a colored band on the plot area covering the range higher than this value. *bandUpperStyle* defines the style of this band.
bandUpperStyle	SVG CSS style definition for the band defined by *bandUpper*.
baseLineStyle	SVG CSS style definition for the base line.
currYAxis	When a chart has multiple y axes, *currYAxis* specifies which yAxis to display on the chart. The chart legend highlights all items that use the current axis. If the user clicks on a legend item that is not highlighted, the chart makes the corresponding axis the active one. See "Chart Axes". Because reports are not interactive, *currYAxis* in a Zen report simply determines the axis that is displayed
gridStyle	SVG CSS style definition. Default style applied to all grid line elements within the plot area for this chart. If defined, *gridStyle* overrides any styles define in the CSS style definition for the page, but *gridStyle* is in turn overridden by any styles defined by a specific axis element in the chart.
labelStyle	SVG CSS style definition. Default style applied to all label elements for this chart. If defined, *labelStyle* overrides any styles define in the CSS style definition for the page, but *labelStyle* is in turn overridden by any styles defined by a specific axis element in the chart.
labelsVisible	If true, display axis labels for this chart (or slice labels for a pie chart). If false, hide labels. The default is true. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
maxLabelLen	The maximum number of characters to display for an axis label. The default value is 20.

Attribute	Description
ongetLabelX	The *ongetLabelX* event handler for the <chart>. Zen invokes this handler to provide the text for labels on the x-axis. See "Zen Component Event Handlers." It must accept an argument, *value*, that contains the 0-based ordinal number of the data point whose text label is to be returned. It may consist of a switch statement based on the value of the input argument. It must return a text string. For example: `ongetLabelX="return zenPage.getXLabels(value);"`
ongetLabelY	Client-side JavaScript expression, as for *ongetLabelX*. For pie charts, *ongetLabelY* is the attribute to use to set slice labels.
ongetSeriesColor	The *ongetSeriesColor* event handler. The chart calls this event handler to get the color for a given data series. The event handler is passed an argument ,*series*, that contains the 0-based ordinal number of the series. If the event handler does not return a color for *series*, the chart uses whatever color is specified by *seriesColors*, *seriesColorScheme*, or *seriesColorsOverride*.
onrenderData	Specifies an optional event handler. If an event handler is defined, it is called by the chart just after it has finished displaying grid lines and data. The event handler is passed an argument *chart* that is the current chart object. It is also passed a *group*, which is the SVG group to which any new SVG content should be added.
onrenderPlotArea	Client-side JavaScript expression. If the chart provides an *onrenderPlotArea* value, then the *onrenderPlotArea* expression is called by the chart just after it displays its underlying plot area (and bands) but before it display grid lines and data. Not supported by Zen reports.
plotAreaStyle	SVG CSS style definition for the plot area panel for this chart.
plotEdgeStyle	SVG CSS style definition applied to the left and bottom edges of the plot area panel for charts that have a 3D appearance.
plotStyle	Default SVG CSS style definition applied to the SVG elements that are used to plot data for this chart. These elements include the line in a line chart, or the bar in a bar chart.
plotToEdge	Specifies how values should be plotted along a category axis: • True— plot the first and last values on the edges of the plot area (as in a line chart) • False— plot values in the centers of each unit (as in a bar chart) This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
scrollButtonStyle	An optional SVG CSS style definition applied to the zoom and scroll buttons. Use the *hasZoom* property to make the zoom and scroll buttons visible.
seriesColors	Comma-separated list of CSS color values to use for data series. The chart applies these colors to each data series in ordinal order (series 0 gets the first color, series 1 the second color, and so on). If you omit *seriesColors*, the chart uses the colors defined by *seriesColorScheme*.

Attribute	Description
seriesColorScheme	This is the name of a built-in color scheme used to plot data series for the chart. Possible values are: “urban”, “tuscan”, “caribbean”, “rustbelt”, “bright”, “glow”, “gray”, “pastel”, and “solid”. The default value is "tuscan".*seriesColors* overrides this property.
seriesColorsOverride	Additional comma-delimited list of CSS color values used for data series. If supplied, this is merged with the colors in the *seriesColorScheme* or *seriesColors* list.
seriesNames	Comma-separated list of names to use for data series. The chart applies these names to each data series in ordinal order (series 0 gets the first name, series 1 the second name, and so on). These names can appear as labels for the series in the legend chart. The *seriesNames* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *seriesNames* string must remain a comma-separated list.
seriesYAxes	Comma-separated list of numbers. These numbers correspond to multiple y axes defined by the chart. The chart uses the axis identified by the number to plot the corresponding data series. The numbers are 0–based. By default, the every data series uses y-axis 0. For example, if `seriesYAxes="1,2,0"`, the chart plots the first data series on second axis, the second data series on the third axis, and the third data series on the first axis. Note that <xyChart>, <diffChart> and <bubbleChart> do not support alternate yAxes. See “Chart Axes”.
stripeStyle	An optional SVG CSS style definition applied to grid stripes when stripes have been enabled with the property *stripesVisible*.
stripesVisible	If true, draw stripes over value axis grid lines. The default value is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”
textSize	Adjusts the size of text used in the chart title and axis labels. Possible values are: ”small”, “medium”, and “large”. If the value is "medium", the chart uses the default font sizes specified by the CSS. if "small" or "large" adjust the size of any text in the chart that does not have an explicit style set by a property.
unselectedItemStyle	An optional SVG CSS style used to indicate unselected chart elements. Used when there is a selected element.
valueBoxStyle	An optional SVG CSS style definition applied to the box that contains the value labels. Use the property *valueLabelsVisible* to make the labels visible.
valueLabelFormat	Specifies a format applied to the numeric values in value labels and tooltips. For a description for this format string, see the section “Format String Field” in the book “Defining DeepSee Models.” Use the property *valueLabelsVisible* to make the labels visible.

Attribute	Description
valueLabelStyle	An optional SVG CSS style definition applied to value labels. This only applies to charts that display element values, such as bar charts. Use the property *valueLabelsVisible* to make the labels visible.
valueLabelsVisible	Specifies whether values should be displayed for elements within the chart. This only applies to charts that display element values, such as bar charts. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

A number of attributes are supported by several chart types. These attributes are listed in the table of attributes for the relevant charts, and described in more detail in the following sections.

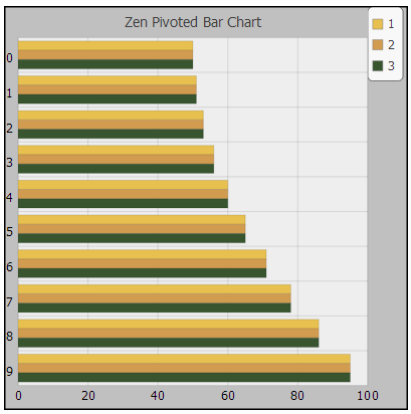
3.3.3.1 chartPivot

If the attribute *chartPivot* is true, rotate the chart so that the x-axis is vertical and the y-axis horizontal. If false, display the chart in typical fashion, with x-axis horizontal and the y-axis vertical.

This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

The following figure shows a pivoted bar chart.

Figure 3–22: Pivoted Bar Chart



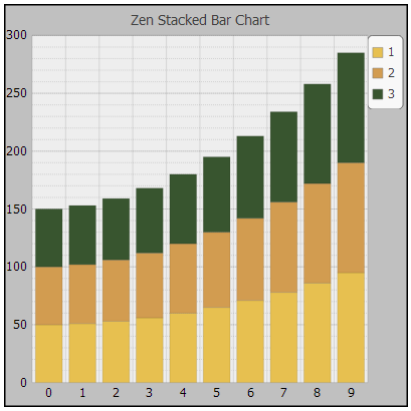
3.3.3.2 chartStacked

The attribute *chartStacked* comes into play if there are multiple data series. If true, the data series are drawn stacked on top of one another (values are additive). If false, the bars appear side by side (the values are independent).

This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

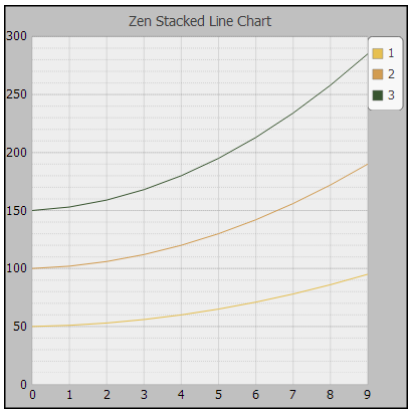
If you set the property *chartStacked* to true, the result is a chart with values at each data point from all the data series in the chart stacked into a single bar. The following figure illustrates a stacked bar chart:

Figure 3–23: Stacked Bar Chart



And the next figure shows the same data presented as a stacked line chart:

Figure 3–24: Stacked Line Chart



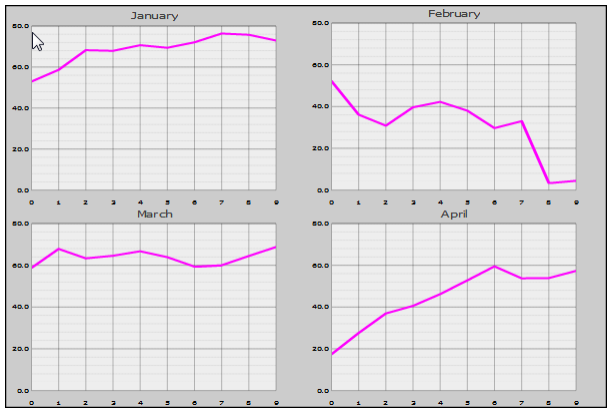
3.3.3.3 showMultiples

If the attribute *showMultiples* is true, charts that support small multiples display the data as a set of small charts, one for each data series. The chart method **hasMultiples** lets you determine whether the chart supports multiples.

This attribute has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”

The following figure shows a line chart with four data series and `showMultiples="true"`.

Figure 3–25: Line Chart Displayed as Multiples



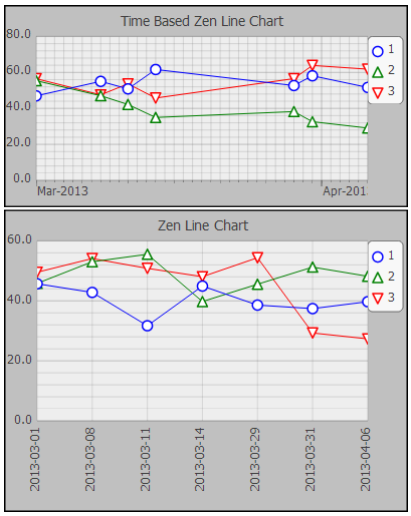
3.3.3.4 timeBased

The attribute *timeBased* specifies that the x-axis is a time line. It is valid only for line and combo chart types. In order to plot a time based chart, you must use the *ongetLabelX* callback method to provide time values in the format YYYY-MM-DD or YYYY-MM-DD HH:MM:SS. These values are placed in their proper position on the x-axis timeline. Because the logic that positions values on the timeline depends on the returned date values, you cannot set the property *maxLabelLen* to a value less than ten.

The chart properties *startTime* and *endTime* set the start and end dates for the time line. If you do not supply values for *startTime* and *endTime*, the chart uses the earliest and latest values returned by the *ongetLabelX* callback method.

The following figure illustrates the effect of a *timeBased* x-axis. The top chart uses a time based x-axis, while the lower chart plots the same x values on a category axis.

Figure 3–26: Time Based Line Chart



3.3.4 Markers

Markers are shapes that are placed at each data point along the chart. If the chart has multiple series, each series in the chart can use a different shape for its marker. Marker attributes apply only to types of chart that support markers (that is, line charts). The base class %ZEN.SVGComponent.chart offers the following marker attributes.

Attribute	Description
markerScale	Decimal value that specifies the scaling for markers. A value of 1.0 (or "") displays markers with the default size. Use a larger value for larger markers, smaller for smaller markers.

Attribute	Description
markerShapes	Comma-separated list of shapes to use for each series. It can be convenient to use a different shape for each series in the chart. This adds further distinction besides the color of each plot in the chart. The list can contain any of the following keywords in any order or combination: • "circle" — a circle • "down" — a triangle with point down • "square" — a square • "up" — a triangle with point up The default value for *markerShapes* is "circle,up,down,square"
markerStyle	SVG CSS style definition. Style applied to all marker elements for this chart. The default is to outline the shape in the same color as the series plot, and fill the shape with white.
markersVisible	If true, and this chart supports markers, markers are displayed for the data points within the chart. If false, no markers are displayed, even if they are defined. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

3.3.5 Legends

The following attributes from the base class %ZEN.SVGComponent.chart determine the style of the chart legend and whether or not it should display.

Attribute	Description
legendHeight	If this chart has a legend, this is the height of the legend box (within the chart coordinate space). You can provide a value as specified in "Specifying Size and Position." The default height is based on the number of data series.
legendLabelStyle	SVG CSS style definition for label text in the legend box.
legendRectStyle	An optional SVG CSS style applied to the rectangle that indicates the current legend in the legend box. Used when the chart has more than one y-axis.
legendStyle	SVG CSS style definition for the background of the legend box.
legendTitle	Title to display in the legend box.
legendVisible	If true, display a legend for this chart. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
legendWidth	If this chart has a legend, this is the width of the legend box. You can provide a value as specified in "Specifying Size and Position." If no *legendWidth* is specified, the chart assigns a default width of 15.

Attribute	Description
legendX	If this chart has a legend, *legendX* provides the x-position of the top left corner of the legend box within the chart coordinate space. You can provide a value as specified in "Specifying Size and Position." If you provide a value relative to the coordinate space, the top left corner of the chart is (0,0) and the bottom right corner is (100,100). The default position for the legend is in the top-right corner of the chart.
legendY	If this chart has a legend, *legendY* provides the y-position of the top left corner of the legend box within the chart coordinate space. Provide values as for *legendX*. The default position for the legend is in the top-right corner of the chart.

3.3.6 Titles

The following attributes from the base class %ZEN.SVGComponent.chart determine the style and contents of the chart title.

Attribute	Description
multipleTitleStyle	An optional SVG CSS style definition used for the title text in each of the small charts created when a data series is displayed as multiple charts. Used when showMultiples is enabled.
subtitle	Subtitle text for the chart. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
subtitleStyle	SVG CSS style definition. Specifies the style for the *subtitle* text.
title	Title text for the chart. By default, it is centered at the top of the chart area. You can use *titleX*, *titleY*, and *titleAlign*. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
titleAlign	Alignment for title and subtitle. Possible values are, "center", "left", and "right". The default value is "center".
titleBoxStyle	SVG CSS style definition. Specifies the style for the box that contains the *title* and *subtitle* text.
titleStyle	SVG CSS style definition. Specifies the style for the *title* text.
titleX	If this chart has a title, this is the x-position of the title. This value is applied as specified by *titleAlign*. For example, if *titleAlign* is "center", *titleX* positions the center of the title. You can provide a value as specified in "Specifying Size and Position." The default is "center".
titleY	If this chart has a title, this is the y-position of the bottom line for the title text. Descenders (for letters such as 'p' and 'q') fall below this line. You can provide a value as specified in "Specifying Size and Position." The default position is just below the top of the chart, within the default *marginTop*.

3.3.7 User Selections

The following attributes control user interactions with the chart. Zen reports does not support any of the properties in this section, because reports are not interactive.

Attribute	Description
onelementClick	The "*onelementClick* event handler" for the chart. Zen invokes this handler whenever the user clicks on a chart element (such as a marker in a line chart, or bar in a bar chart). See "Zen Component Event Handlers." It must accept an argument, *chart*, that represents this chart object. It then calls a method on this object to determine the identity of the currently selected data point (**getSelectedItem**) or data series (**getSelectedSeries**). Either method returns the 0–based ordinal number of the item that was selected, or -1 if nothing is currently selected. For example: `onelementClick="zenPage.chartElementClick(chart);"` If no *onelementClick* expression is provided, Zen uses its own handler to provide values for *selectedItem* and *selectedSeries*. Not supported by Zen reports.
selectedItem	0–based ordinal number of the currently selected chart element (such as a marker in a line chart, or bar in a bar chart). This value is -1 if nothing is currently selected. Not supported by Zen reports.
selectedItemStyle	SVG CSS style definition for the currently selected item in the chart. Not supported by Zen reports.
selectedSeries	0–based ordinal number of the currently selected series in the chart, or -1 if no series is currently selected. Not supported by Zen reports.

3.4 Chart Axes

Zen displays all charts except pie charts with x and y axes. Axes determine how chart data is displayed. You can use <xAxis> or <yAxis> elements to specify the range of axis values (*minValue* to *maxValue*), or Zen can determine these values automatically based on the range of data values.

- <xAxis> can be a *category axis* or a *value axis*, depending in the type of chart. A category axis names the data categories. A value axis indicates the values of plotted data.
- <yAxis> is a *value axis*.

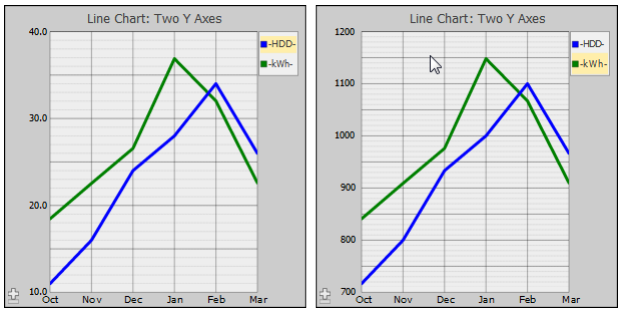
You can supply more than one y-axis to a chart, which enables you to plot data series with different ranges of values on the same chart in a meaningful way. The following example plots six months of data on heating degree days (HDD), and electricity use in kilowatt hours (kWh) for a residential building in the Boston MA area. A heating degree day is a measure of the energy demand for heating buildings. In this example, data values for HDD range from 11 to 34, and data values for kWh range from 841 to 1148. The following code fragment creates a line chart that plots these two data series using two different axes, one for values from 10 to 40, the other for values from 700 to 1200:

XML

```
<lineChart id="chart"
  ongetData="return zenPage.getLineChartData(series);"
  onelementClick="zenPage.chartElementClick(chart);"
  ongetLabelX="return zenPage.getXLabels(value);"
  seriesNames="HDD,kWh"
  backgroundStyle="fill: #cccccc;"
  plotAreaStyle="fill: #eeeeee;"
  title="Line Chart: Two Y Axes"
  currYAxis="1"
  seriesCount="2"
  seriesSize="6"
  seriesColorScheme="solid"
  plotStyle="stroke-width: 1px;"
  labelsVisible="true"
  seriesYAxes="1,0"
  width="300"
  height="300">
  <xAxis id="xAxis" />
  <yAxis minorGridLines="true" minValue="700" maxValue="1200" labelUnits="100"
    minorUnits="10" majorUnits="50" majorGridLines="true" />
  <yAxis minorGridLines="true" minValue="10" maxValue="40" labelUnits="10" baseValue="0"/>
</lineChart>
```

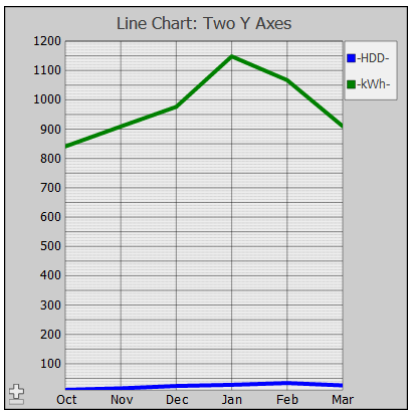
The property *seriesYAxes* associates data series with axes. The property *currYAxis* specifies the axis that is active when you first display the chart. See "Plot Area". The following figure shows the resulting chart. The first image shows the chart as it is initially displayed, with the HDD axis visible. The HDD item in the legend is highlighted to show that it is the relevant data series for the axis currently in use. If you click on kWh in legend, the chart displays the appropriate y-axis. All data series associated with an axis are all highlighted when that axis is active.

Figure 3–27: Line Chart with Two y Axes



Compare these charts with the following chart, which attempts to plot both HDD and kWh on a single axis.

Figure 3–28: Same data Plotted on One y-axis



Note that <xyChart>, <diffChart> and <bubbleChart> do not support alternate yAxes.

The following attributes from the class %ZEN.Auxiliary.axis are available as attributes of either <xAxis> or <yAxis> within a chart definition. All of these attributes are optional; Zen provides reasonable defaults for each of them based on the data supplied to the chart.

Table 3–2: Chart Axis Attributes

Attribute	Description
axisType	Provides additional control over the display of labels for this axis. If the value is a null string ("") the axis shows a value or category label. If the value is "percent" the axis shows a value label as a percentage. *axisType* also affects the format of the tooltip for values plotted on this axis. If *valueLabelFormat* is also set, that format takes precedence for tooltips.
baseValue	For charts with filled regions (bar charts), *baseValue* is a decimal value that specifies where the base of the filled region should be plotted. If missing or blank (""), the base is the bottom of the plot area.
labelPosition	Specifies the side of the chart where the labels for this axis appear. Possible values for a y-axis are "left" and "right", and for an x-axis are "top" and "bottom". The default value is “left” for a y-axis and “bottom” for an x-axis.
labelStyle	SVG CSS style definition for the text labels along this axis.
labelUnits	Decimal value that specifies the amount of space between labels along a value axis. Ignored by category axes. If *labelUnits* is missing or blank (""), Zen automatically calculates a value based on the data series.
majorGridLines	If true, grid lines are displayed for each major unit on this axis. If false, major grid lines are not displayed. The default is true. This attribute has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”
majorGridStyle	SVG CSS style definition for the major grid lines along this axis.
majorUnits	Decimal value that specifies the amount of space between major grid lines along this axis. If *majorUnits* is missing or blank (""), Zen automatically calculates a value based on the data series.
maxValue	Decimal value that specifies the maximum data value along this axis. If *maxValue* is missing or blank (""), Zen automatically calculates a value based on the data series.
minValue	Decimal value that specifies the minimum value along this axis. If *minValue* is missing or blank (""), Zen automatically calculates a value based on the data series.
minorGridLines	If true, grid lines are displayed for each minor unit on this axis. If false, minor grid lines are not displayed. The default is true. This attribute has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”
minorGridStyle	SVG CSS style definition for the minor grid lines along this axis.
minorUnits	Decimal value that specifies the amount of space between minor grid lines along this axis. If *minorUnits* is missing or blank (""), Zen automatically calculates a value based on the data series.

Attribute	Description
title	Title text for the axis. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

When you work with %ZEN.SVGComponent.chart subclasses programmatically, you work with axes as the xAxis and yAxis properties of the chart object. Each of these properties is a %ZEN.Auxiliary.axis object with properties corresponding to the attributes listed above.

4

Zen Forms

Forms permit the user to enter data. A Zen *control* is a component that displays application data and allows the user to edit this data. A Zen *form* is a specialized group component designed to contain control components. Zen forms have the same style and layout attributes as any Zen group. Also, because they are groups, forms may contain any other type of Zen component.

Like all Zen components, a Zen form must be the child of a Zen page object. This means that if you want to provide a form for a Zen application, you must create a Zen page class that includes a form component inside the <page> container. Two components are available:

- "<form>" — A Zen group that contains a specific list of control components. These controls may or may not take their values from a data controller, but their layout is entirely determined by the <form> definition in XData Contents.
- "<dynaForm>" — An extension of <form> that dynamically injects control components into a group (or groups) on the Zen page. The list of controls may be determined by the properties of an associated data controller, or by a callback method that generates a list of controls. Layout is automatic, determined by code internal to the <dynaForm>.

There is a Studio tutorial that uses Zen wizards to create a form-based user interface for a simple application. See the chapter "Building a Simple Application with Studio" in the book *Using Studio*.

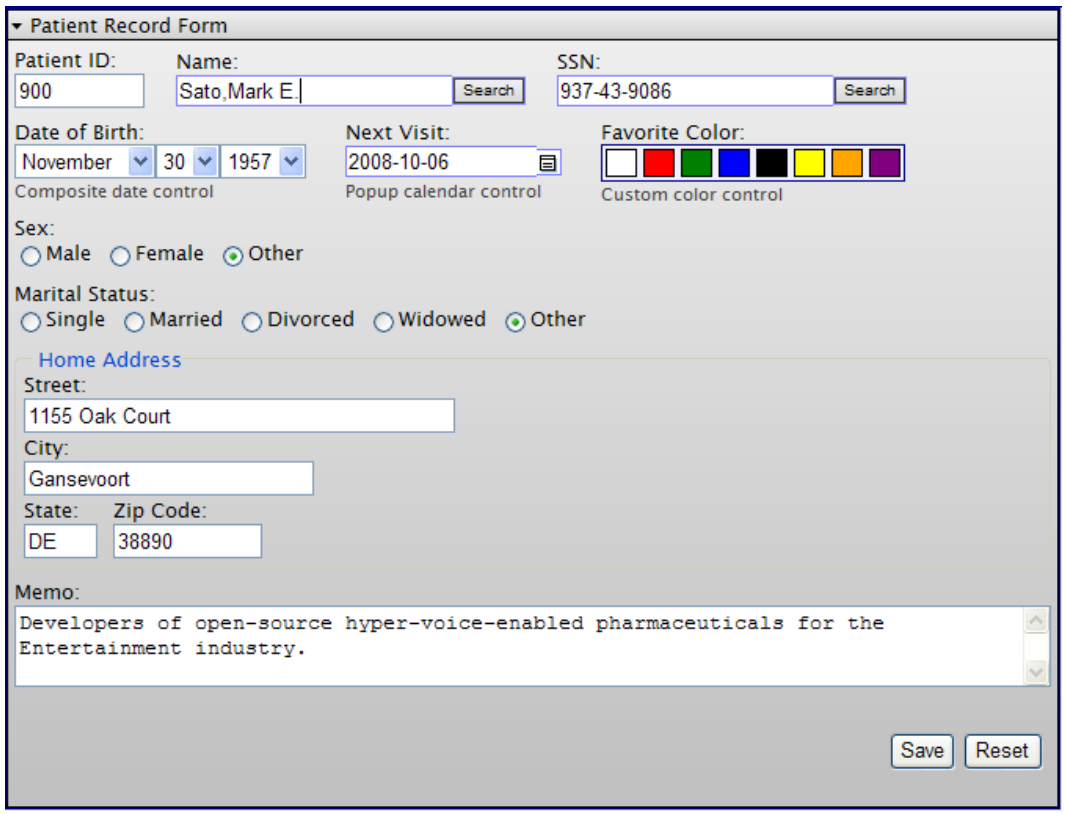
For information about data controllers, see the chapter "Model View Controller."

Chapter topics incluide:

- "Forms and Controls"
- "User Interactions"
- "Defining a Form"
- "Providing Values for a Form"
- "Detecting Modifications to the Form"
- "Validating a Form"
- "Errors and Invalid Values"
- "User Login Forms"
- "Dynamic Forms"

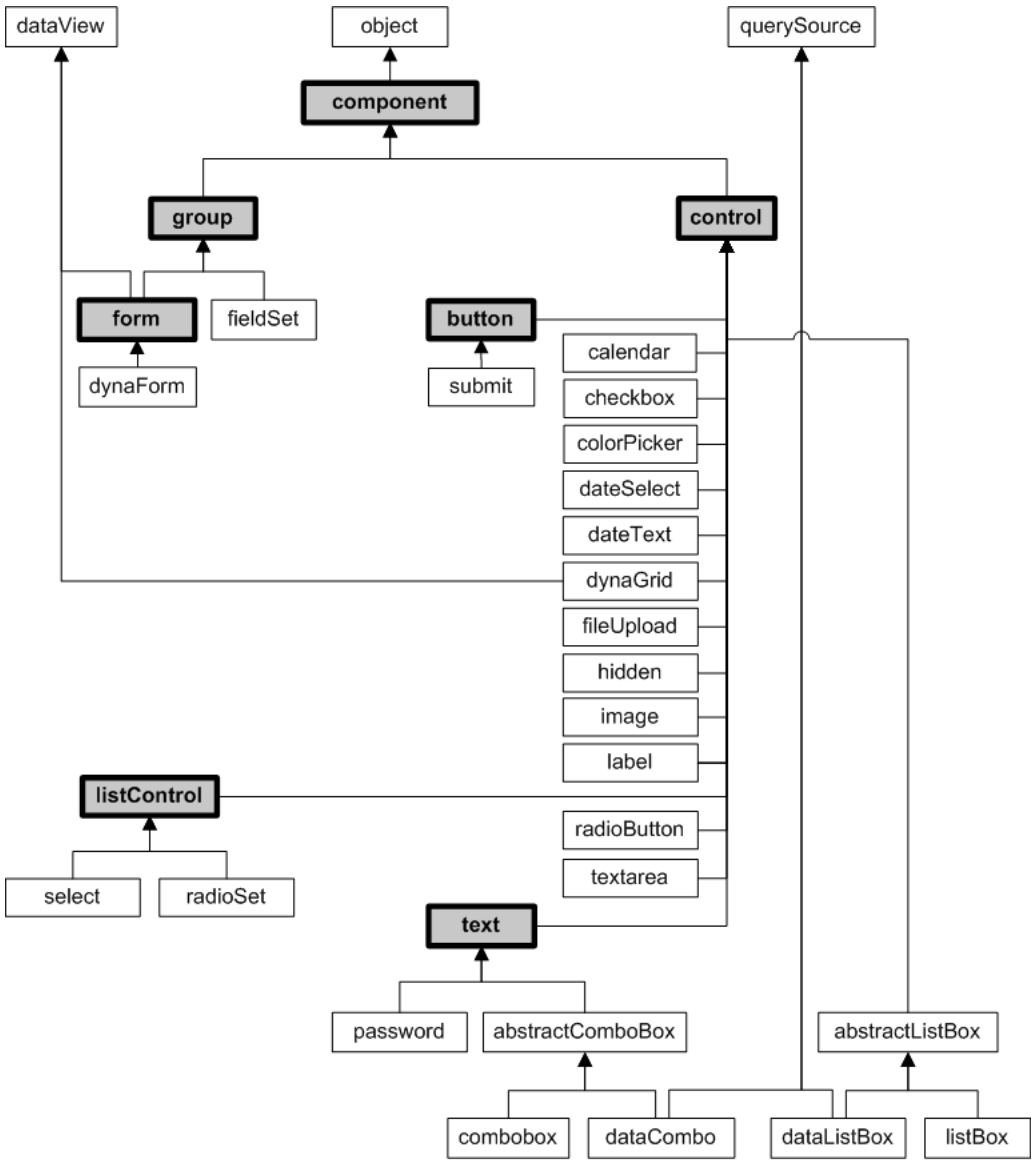
4.1 Forms and Controls

The Zen library includes a number of controls for use in forms. Many of these controls are wrappers for the standard HTML controls, while others offer additional functionality. The following figure displays a form that contains a number of controls, including text fields and radio buttons. This is the sample form generated by the class ZENDemo.FormDemo in the SAMPLES namespace.



The following figure lists the form and control components that Zen provides. Most of the classes shown in the figure are controls. All of the classes shown in the diagram are in the package %ZEN.Component, for example %ZEN.Component.form and %ZEN.Component.control. The diagram shows the inheritance relationships for these classes, and highlights the base classes most frequently discussed in this book.

Figure 4–1: Class Inheritance Among Form and Control Components



For details about individual controls, see the chapter "Zen Controls."

4.2 User Interactions

The basic interaction between a Zen application user and a Zen form is as follows:

1. A user interacts with controls on the form
2. Zen may validate the data as it is entered
3. A user action indicates that it is time to submit the form
4. Zen may validate the data prior to attempting the submit
5. Zen interacts with the user to handle any errors it finds
6. When all is well, Zen submits data from the form

7. Any of the following might happen next:
   - Data from the form may be written to the server
   - The same Zen page may redisplay
   - A different Zen page may display
   - The same Zen page may display, but with components added or changed

4.3 Defining a Form

The Zen components "<form>" and "<dynaForm>" each support the following attributes for defining the characteristics of a form.

Table 4–1: Form Component Attributes

Attribute	Description
Zen group attributes	A form has the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
action	Specifies a HTML *action* for the form. Setting the *action* attribute overrides the default behavior of Zen forms so that Zen does not execute its normal submit logic. InterSystems recommends you do not use the *action* attribute except in special cases where direct control of the HTML *action* attribute is required. This could be the case for certain custom login forms.
autocomplete	Indicates whether controls in this form can have their values automatically completed by default by the browser. Elements belonging to the form can override this setting by setting an autocomplete attribute.
autoValidate	If true (the default), automatically invoke this form's **validate** method whenever this form is submitted. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
controllerId	If this form is associated with a data controller, the *controllerId* attribute identifies the controller that provides the data for this form. The *controllerId* value must match the *id* value provided for that <dataController>. See the chapter "Model View Controller."
enctype	Specifies a HTML *enctype* for the form, such as "multipart/form-data"
invalidMessage	Message text that the form validate method displays in an alert box when the contents of this form are invalid. The default is: `This form contains invalid values. Please correct the following field(s) and try again.` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

Attribute	Description
key	String that identifies a specific instance of the data model object that is associated with this form. This is the model ID. The format and possible values of the model ID are determined by the developer of the data model class. For details, see "Data Model Properties" in the chapter "Model View Controller." If a <form> or <dynaForm> specifies a *key*, the *OnLoadForm* callback uses this model ID to load initial values into the form. However, if this form is connected to a <dataController>, the *key* value is ignored. The *key* can be a literal string, or it can contain a Zen #()# runtime expression.
method	Specifies a HTML *method* for the form. Setting the *method* attribute overrides the default behavior of Zen forms so that Zen does not execute its normal submit logic. InterSystems recommends you do not use the *method* attribute except in special cases where direct control of the HTML *method* attribute is required. This could be the case for certain custom login forms.
nextPage	URI of the page to display after this form is successfully submitted. This URI may be overwritten by a specific <submit> button on the form.
onchange	The *onchange* event handler for the form. Zen invokes this handler when the value of a control on this form is changed by the user or when the modified flags are cleared by a call to the form's **clearModified** method. When fired for a control, the *onchange* expression can use an argument called `control` to reference the modified control. See "Zen Component Event Handlers."
ondefault	Client-side JavaScript expression that Zen invokes when the user performs an action that triggers the default action for a form. Typically this is when the user presses the **Enter** key within a control within the form.
oninvalid	Client-side JavaScript expression that Zen invokes when this form's **validate** method determines that the contents of this form are invalid. This provides the application with a chance to display a custom message.
OnLoadForm	Name of a server-side callback method in the Zen page class. Find further information following this table.
onnotifyView	The *onnotifyView* event handler for the form. Zen invokes this handler each time the data controller connected to this form raises an event. See "Zen Component Event Handlers."This attribute applies if the form is associated with a data controller. See the chapter "Model View Controller."
onreset	Client-side JavaScript expression that Zen invokes when this form is about to be reset. Generally this expression invokes a client-side JavaScript method.

Attribute	Description
OnSubmitForm	Name of a server-side callback method in the Zen page class. This method takes a set of actions that is appropriate upon form submit. Use of *OnSubmitForm* is limited to providing an alternative to the built-in mechanisms that a form provides for detecting changes, validating values, and submitting the form. It should not be used to perform other types of general-purpose processing. The next several sections describe the build-in mechanisms provided by form submit. Zen invokes this method when the user submits the form, automatically passing it an input parameter of type **%ZEN.Submit**. If no *OnSubmitForm* value is specified for the form, Zen invokes the page's **%OnSubmit** method instead. To follow this sequence, see the section "Processing a Form Submit." The callback must return a %Status data type. The following is a valid signature for this callback: `ClassMethod SubmitForm(pSubmit As %ZEN.Submit) As %Status` To use the above method as the callback, the developer would set `OnSubmitForm="SubmitForm"` for the <form> or <dynaForm>.
onsubmit	Client-side JavaScript expression that Zen invokes when this form is about to be submitted. Generally this expression invokes a client-side JavaScript method with a Boolean return value. Invoking this method gives Zen a chance to perform client-side validation of values within the form. If the method returns false, the pending submit operation does not occur. Note that unlike the HTML onsubmit event, the *onsubmit* callback is always called whenever the form is submitted.
onvalidate	Client-side JavaScript expression that Zen invokes when this form's **validate** method is called. Generally this expression invokes a client-side JavaScript method that performs validation.
readOnlyMessage	Message text that the form validate method displays in an alert box when the user makes an attempt to save a form bound to a read-only data model. The default *readOnlyMessage* text is: `This data is read only.` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

The *OnLoadForm* method retrieves the values that appear on the form when it first displays. It can get values from the object whose model ID is provided by the *key* attribute for the form, or it can assign literal values. The method must then place these values into the input array, subscripted by the *name* attribute for the corresponding control on the form. It is this *name* (and not the *id*) that associates a control with a value. Zen invokes this method when it first draws the form, automatically passing it the following parameters:

- %String — the *key* value for the form
- An array of %String passed by reference

The callback must return a %Status data type. The following example shows a valid method signature and use of parameters:

Class Member

```
Method LoadForm(pKey As %String,
                ByRef pValues As %String) As %Status
{
  Set emp = ##class(ZENDemo.Data.Employee).%OpenId(pKey)
  If ($IsObject(emp)) {
    Set pValues("ID") = emp.%Id()
    Set pValues("Name") = emp.Name
    Set pValues("SSN") = emp.SSN
  }
  Quit $$$OK
}
```

To use the above method as the callback, the developer would set `OnLoadForm="LoadForm"` for the <form> or <dynaForm>.

4.4 Providing Values for a Form

A form can initially display with blank fields, or you can provide data for some of the fields. Providing data for a field that the user sees means setting the value property for the associated Zen control component, before you ask Zen to display the form. There are several ways to do this:

- Set the *value* attribute while adding each control to XData Contents.

  ```
  <text value="hello"/>
  ```

- Set the *onLoadForm* attribute while adding the form to XData Contents.

  ```
  <form id="MyForm" OnLoadForm="LoadForm">
  ```

- Set value properties from the page's **%OnAfterCreatePage** method.

  ```
  Do ..%SetValueById("Doctor",$G(^formTest("Doctor")))
  ```

- On the client side, call the **setValue** method for each control.

Presumably, once the user begins editing the form, any initial values may change.

4.5 Detecting Modifications to the Form

A Zen form component tracks whether or not changes have occurred in any control on the form. %ZEN.Component.form and %ZEN.Component.control each offers a client-side method called **isModified** that tests this programmatically.

A control's **isModified** method returns true if the current logical value of the control (the *value* property) is different from its original logical value (the *originalValue* property). With each successive submit operation, the *originalValue* for each control acquires its previous *value*, so that this answer is always current with respect to the current state of the form.

When you call a form's **isModified** method, it invokes **isModified** for each control on the form, and if any control returns true, **isModified** returns true for the form.

Note: The <textarea> control returns an accurate **isModified** status when it contains fewer than 50 characters. When the <textarea> value contains 50 characters or more, the control does not compute an **isModified** status.

4.6 Validating a Form

Each Zen form has a **validate** method whose purpose is to validate the values of controls on the form. If the form's autoValidate property is true, **validate** is called automatically each time the form is submitted. Otherwise, **validate** may be called explicitly by the application. **validate** does the following:

1. Calls a form-specific onvalidate event handler, if defined. If this event returns false, Zen declares the form invalid and no further testing occurs.
2. Resets the invalid property of all the form's controls to false, then tests each control by calling the control's **validationHandler** method. This method, in turn, does the following:
   - If the control's readOnly or disabled properties are true, return true.
   - If the control's required property is true and the control does not have a value (its value is ""), return false.
   - If the control defines an onvalidate event, execute it and returns its value. Otherwise, call the control's **isValid** method. **isValid** can be overridden by subclasses that wish to provide built-in validation (such as the dateText control).
3. As the **validate** method tests each control, the form builds a local array of invalid controls.
4. After the **validate** method is finished testing the controls, it returns true if the form is valid.
5. If the form contains one or more controls with invalid values, it is invalid. **validate** performs one of the following additional steps to handle this case:
   - If the form defines an oninvalid event handler:

     Execute the handler. This provides the form with a chance to handle the error conditions. The value returned by the oninvalid event is then returned by the form's **validate** method. The oninvalid handler has an argument named *invalidList* that receives a JavaScript array containing the list of invalid controls. For example:

     XML

     ```
     <form oninvalid="return zenPage.formInvalid(zenThis,invalidList);" />
     ```

     Where the **formInvalid** method looks like this:

     Class Member

     ```
     ClientMethod formInvalid(form,list) [ Language = javascript ]
     {
       return false;
     }
     ```

   - When the form has no oninvalid event handler:

     validate sets the invalid property to true for each invalid control (which changes their style to zenInvalid); gives focus to the first invalid control; and displays an error message within an alert box. The message displayed in the alert box is built from a combination of the form's invalidMessage property along with the value returned from each invalid control's **getInvalidReason** method.

Note: It is standard practice not to invoke validation logic for null values.

4.7 Errors and Invalid Values

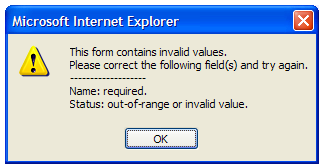
Should an error occur while a form is being submitted, or should the form fail validation, Zen redisplays the page containing the form. The user has the opportunity to re-enter any incorrect values and submit the page again. All of this is extremely easy to set up when adding the <form> or <dynaForm> component to the Zen page.

The SAMPLES namespace class ZENTest.FormTest allows you to experience error handling with Zen forms as follows:

1. Start your browser.
2. Enter this URI:

   ```
   http://localhost:57772/csp/samples/ZENTest.FormTest.cls
   ```

   Where 57772 is the web server port number that you have assigned to Caché.
3. Ensure that the **Name** field is empty.
4. Clear the **Status** check box.
5. Click **Submit**.
6. The form's **validate** method detects the invalid fields and highlights them with pink.
7. The following alert message displays in the browser.

   To generate this message, the form has assembled the following snippets of text. Zen offers default values for these items, so there is no need for you to do anything for the default message to appear. However, if you wish you can customize them to any extent:

   - The message begins with the form's *invalidMessage* text.
   - For each control that has its *required* attribute set to true, but contains no entry, the message lists the control's *label* followed by the control's *requiredMessage* text.
   - For each control that contains an invalid value, the message lists the control's *label* followed by the control's *invalidMessage* text.
8. Click **OK** to dismiss the alert message box.
9. The invalid controls remain highlighted until the user edits them and resubmits the form.

4.8 Processing a Form Submit

The request to submit the contents of a form can be triggered in one of two ways:

- The user clicks a "<submit>" button placed within the form.
- The application calls the **submit** method of the form object in response to a user event:

  ```
  %form.submit
  ```

When a form is submitted, the values of the controls in the form are sent to the server and the **%OnSubmit** callback method of the page containing the form is called. Note that the **%OnSubmit** callback method is that of the page that contains the form, not of the form itself or of any component on the form.

%OnSubmit receives an instance of a special %ZEN.Submit object that contains all the submitted values. Note that there is no page object available during submit processing. Zen automatically handles the full details of the submit operation, including invoking server callbacks and error processing. All forms are submitted using the HTTP POST submission method.

Note that using the setting ENCODED=2 disables the <form> component, because ENCODED=2 removes all unencrypted parameters from the url.

If you are interested in details of how Zen executes a submit operation, the following table lists the internal events in sequence, organized by user viewpoint, browser-based execution, and server-side execution. Most of the communication details are handled by the CSP technology underlying Zen. For background information, see the "Zen Client and Server" chapter in *Using Zen*.

Table 4–2: Form Submit Sequence

	User Viewpoint	In the Browser	On the Server
1	User clicks a button to invoke form submit		
2		Post control values to the server via the HTTP POST mechanism	
3			Deserialize data from the client
4			Reconstruct DOM based on new control values
5			Run the server-side code for the page
6			Update server-side DOM
7			Generate new HTML page
8			Send new HTML page as response to HTTP POST
9		Render the HTML received in HTTP POST response	
10		Update client-side DOM to reflect changes made on the server	
11	User sees new page		

4.9 User Login Forms

An application login page presents a special case of a Zen form. To create application login and logout pages, see the section "Controlling Access to Applications" in the "Zen Security" chapter of *Developing Zen Applications*.

4.10 Dynamic Forms

A <dynaForm> is a specialized type of form that dynamically injects control components into a group (or groups) on the Zen page. Layout is determined automatically by code internal to the <dynaForm>. The list of controls may be determined by the properties of an associated data controller, or by a callback method that generates a list of controls.

<dynaForm> has the following attributes:

Attribute	Description
Form component attributes	A <dynaForm> has the same general-purpose attributes as any Zen form. For descriptions, see the section "Defining a Form." The general-purpose form attributes include the *controllerId* and *onnotifyView* attributes needed to work with a data controller.
controllerId	If this form is associated with a data controller, the *controllerId* attribute identifies the <dataController> component that provides the data for this form. The *controllerId* value must match the *id* value for the <dataController>. For full details about creating a <dynaForm> that uses a data controller, see the chapter "Model View Controller."
defaultGroupId	The *id* of a group in which to place the controls generated by this <dynaForm>. This provides a way to control layout. Somewhere inside the <dynaForm>, you must specify a group component (such as <vgroup> or <hgroup>) with an *id* that matches the *defaultGroupId*. If no *defaultGroupId* is provided, controls are added directly to the <dynaForm> without being contained in a group.
injectControls	By default, Zen places any automatically inserted controls *after* (that is, below or to the right of) any manually inserted controls on the <dynaForm>. If you want automatically inserted controls to appear before (that is, above or to the left of) any manually inserted controls, set the *injectControls* attribute to `"before"` as in the following example. This code causes the controls provided by the data controller to appear `"before"` the manually inserted **Save** button: The possible values for *injectControls* are `"before"` and `"after"`. The default is `"after"`.
OnGetPropertyInfo	Name of a server-side callback method in the Zen page class. This method prepares additional controls to inject onto the form when it is displayed. Zen invokes this method when it first draws the form. See the discussion following this table.
onnotifyView	The *onnotifyView* event handler for the form. This attribute applies if the form is associated with a data controller. Zen invokes this handler each time the data controller connected to this form raises an event. See "Zen Component Event Handlers." For full details about creating a <dynaForm> that uses a data controller, see the chapter "Model View Controller."

The callback method identified by the *OnGetPropertyInfo* attribute prepares additional controls to inject onto the form when it is displayed. If this method is defined in the page class, Zen invokes it when it first draws the form, automatically passing it the following parameters:

- %Integer — the next index number to apply to a control on the generated form. As the callback method injects additional controls on the form, it must increment this index value, as shown in the example below.

- As array of %String passed by reference.
- %String — the current data model ID, for cases where the contents of a dynamic form vary by instance of the data model object. The method can get values from this object, or it can assign literal values.

To define additional controls, the method must place values into the input array, using two-part subscripts as follows:

1. The first subscript is the value of the *name* attribute that was assigned to the corresponding control on the form. It is this *name* (and not the *id*) that identifies the control and associates it with a value. For example, the following array position stores the 1–based index of the control's ordinal position on the form:

   `pInfo(`*name*`)`

2. The second subscript is the name of a property on the control object. `%type` is a control property whose value identifies the type of the control. The names of other properties are listed in the "Zen Controls" chapter, either in the "Control Attributes" section or in topics about specific controls. For example, the following array position stores the value for the *attr* attribute of the *name* control:

   `pInfo(`*name*`,`*attr*`)`

The callback must return a %Status data type. The following example shows a valid method signature and use of parameters and array subscripts:

Class Member

```
Method GetInfo(pIndex As %Integer,
               ByRef pInfo As %String,
               pModelId As %String) As %Status
{
  Set pInfo("Field1") = pIndex
  Set pInfo("Field1","%type") = "textarea"
  Set pInfo("Field2") = pIndex + 1
  Set pInfo("Field2","label") = "Field 2"
  Quit $$$OK
}
```

To use the above method as the callback, the developer would set `OnGetPropertyInfo="GetInfo"` for the <dynaForm>.

If the last control on the generated form comes from an embedded property with *n* fields, your callback must increment past these generated controls by adding *n* to the index number at the beginning of the method, before assigning the index to any controls. This convention is not necessary when the last generated control on the form comes from a simple data type, such as %String, %Boolean, or %Numeric. The previous example shows the simple case, in which the method uses and increments the index provided.

5

Zen Controls

Zen controls are the user input elements that you place on a Zen form. All controls are Zen components, but Zen controls also have unique characteristics derived from their parent class %ZEN.Component.control.

Most importantly, each control has a value associated with it. value is a property that contains the current logical value of the control. Each control has the ability to display this value or keep it internally. Zen validates and submits all the control values for a form together, as a unit, according to the rules described in the chapter "Zen Forms."

This chapter:

- Describes characteristics shared by all Zen controls:
  - "Control Attributes"
  - "Data Drag and Drop"
  - "Control Methods"
- Identifies variations in layout, style, and behavior for different categories of control:
  - "Buttons" — <button>, <image>, <submit>
  - "Text" — <label>, <text>, <textarea>, <password>
  - "Selections" — <checkbox>, <multiSelectSet>, <fileUpload>, <colorPicker>, <radioButton>, <radioSet>
  - "Lists" — <select>, <listBox>, <dataListBox>, <combobox>, <dataCombo>
  - "Dates" — <calendar>, <dateSelect>, <dateText>
  - "Grid"— <dynaGrid>, <dataGrid>
  - "Hidden" — <hidden>

5.1 Control Attributes

All Zen controls have the following attributes in common.

Table 5–1: Control Component Attributes

Attribute	Description
Zen component attributes	A control has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen* Of these attributes, *name* has special significance for a control. When the form is submitted, your code must use this *name* (and not the *id*) to retrieve the value of the control. If the control does not have a *name*, its value cannot be retrieved. To avoid clashing with *name* values reserved by Caché Server Pages, do not use any *name* for a Zen control that begins with the string `Cache`. Also avoid using punctuation characters in *name* values, particularly the _ underscore character. For more information that applies to all control components, see "Data Drag and Drop" and "Control Methods" in this chapter.
clientType	Indicates the client-side (JavaScript) data type to expect for this control's *value*. By default, a controls treats its *value* as a string with no client-side normalization. However, a control can set a value for *clientType* to indicate that it has a non-string value on the client side. Possible values are: • `"string"` — The client-side *value* is a string. • `"boolean"` — The client-side *value* is true or false. • `"integer"` — The client-side *value* is either an integer or `''` to indicate an invalid integer. • `"float"` — The client-side *value* is either a float or `''` to indicate an invalid float.
controlClass	Name of a CSS style class. When Zen lays out this control, it assigns this value to the primary HTML element displayed for this control.
controlStyle	String containing a CSS style definition. Zen applies this style to the primary HTML element displayed for this control.
dataBinding	If this control is associated with a data controller, this attribute identifies the specific property within the <dataController> *modelClass* that provides the value for this control. See the chapter "Model View Controller."
disabled	If true, this control is disabled; its appearance is unchanged, but it does not respond to user actions. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
invalid	Set to true when the value of this control is known to be invalid. Zen form validation logic does this so that Zen can display this control in a way that indicates its value is invalid. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

Attribute	Description
invalidMessage	Message text to provide when *invalid* is true. The default is: `out-of-range or invalid value.` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
—	The next several attributes in this table (names beginning with "on...") identify event handlers for user actions relating to a control. Zen invokes the handler when the related event occurs. See "Zen Component Event Handlers."
onblur	Fired when the control loses focus.
onchange	Fired when the value of the control changes. Note that controls fire this event indirectly; the actual onchange event is sent to a built-in handler that notifies the form that owns this control of the modification.
onclick	Fired when the mouse is clicked on the control.
ondblclick	Fired when the mouse is double-clicked on the control.
onfocus	Fired when the control is given focus.
onkeydown	Fired when the user presses down on a key while this control has focus. There is a corresponding *onkeyup* for when the user releases the key.
onkeypress	Fired after the user has pressed a key (that is, the user has pushed down and then released the key) while this control has focus.
onkeyup	Fired when a key is released while this control has focus.
onmousedown	Fired when a mouse button is released while within the area of the control.
onmouseout	Fired when the mouse pointer leaves the area of the control.
onmouseover	Fired when the mouse pointer enters the area of the control.
onmouseup	Fired when a mouse button is pressed while within the area of the control.
onsubmit	Fired when the form this control belongs to is submitted. This gives controls a chance to supply or modify the value they submit.
onvalidate	Fired when this control's value is validated by its parent form.
—	(End of the list of attributes that identify event handlers.)
originalValue	Original value for this control before any user modification. It is used to detect which controls have been modified. This is a special value in that it is automatically initialized when a form is displayed. On the client side, do not access this property directly; instead use the **getProperty** and **setProperty** client-side methods. Note that setting the *originalValue* property on the client (via **setProperty**) resets it to the current *value* of this control.

Attribute	Description
readOnly	If true, this control is read-only. The user cannot change the value of this control, but the control still appears on the form and still submits its value when the form that contains it is submitted. Setting *readOnly* to true effectively disables the component; this is the standard HTML behavior for select controls. The default *readOnly* value is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
required	If true, this control is required. That is, a user must supply a value for this control or the default form validation logic fails. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
requiredMessage	Message text that the form validate method displays in an alert box when this control is required and does not have a value. The default *requiredMessage* text is: `required.` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
tabIndex	Integer used to provide a value for the HTML *tabIndex* attribute. The browser uses this attribute to control the tab order of controls within a form.
value	Default value displayed within this control. This is a special value in that it is automatically initialized when a form is displayed. The *value* can be a literal string, or it can contain a Zen #()# runtime expression. On the client side, do not access this property directly; instead use the **getValue** and **setValue** client-side methods.

5.2 Data Drag and Drop

All Zen controls support data drag and drop functionality as follows:

- Data drag and drop features work only if the containing <page> has its *dragAndDrop* attribute set to true.
- A *data drag* from a control occurs when the user clicks the mouse button down while the cursor is positioned on the control, then moves the mouse away from the control while still holding down the button. Data drag captures the current value of the control where the drag operation began. If the logical value and displayed value are different, this difference is preserved when the data is captured. If a control has multiple values, such as a list that presents several items, then the captured value is the value of the list item where the cursor was positioned when the user clicked the mouse button. Data drag works only if the control where the drag began has its *dragEnabled* attribute set to true.
- A *data drop* onto a control occurs when the user releases the mouse button following a drag operation from another control. Data drop replaces the logical value and displayed value of the control where the drop occurred with the values that were being dragged. If a control has multiple values, such as a list that presents several items, the "drop" operation drops the value onto the specific list item where the cursor was positioned when the user released the mouse button. Data drop works only if the control where the drop ended has its *dropEnabled* attribute set to true.

For more information about how to configure data drag and drop for Zen controls, consult the following sections:

- "Drag and Drop" in the "Zen Component Concepts" chapter of *Using Zen* introduces all types of drag and drop operations. It explains how to enable and configure data drag and drop features for controls while placing them on a Zen page. Briefly, you can enable the features by setting the *dragEnabled* and *dropEnabled* attributes to true. Having done that, you have the option of configuring the exact ways in which these features work, by setting the *onafterdrag*, *onbeforedrag*, *ondrag*, and *ondrop* attributes. Special configuration is not necessary, as each control has its own way of handling drag and drop, but it is available if you prefer it.
- "Data Drag and Drop Methods" in the "Custom Components" chapter of *Developing Zen Applications* explains how to write a custom control component that has unique drag and drop behavior. This ensures that, if you want special drag and drop behavior for a control, the desired behavior is applied consistently each time a developer places the control on a Zen page, without requiring the developer to configure the control by setting the *onafterdrag*, *onbeforedrag*, *ondrag*, and *ondrop* attributes.
- "<listBox> Drag and Drop" in the "Lists" section of this chapter explains how the <listBox> control allows the user to reorder entries in a list, or move an entry from one list to another, using drag and drop motions. This is in addition to data drag and drop, which the <listBox> also supports.
- For the most part, data drag and drop applies only to control components, each of which has a logical value and a display value. However, <dynaTree>, which is not a control, supports data drag because each of its nodes actually has a logical value and a display value. For details, see "<dynaTree> Drag and Drop" in the chapter "Navigation Components."

5.3 Control Methods

Each control has internal methods, which you can study in more detail by viewing the online class documentation for %ZEN.Component.control and its subclasses.

The most important of these methods are those used to manipulate the value of the control. You can get and set the value property using the client-side **getProperty** and **setProperty** methods. In fact, **getProperty** and **setProperty** are available for you to work programmatically with any of the properties listed in the "Control Attributes" section, not just value. Calling the **setProperty** method ensures that all actions relating to setting the property also occur, including rendering the content within the control on the page.

Zen controls also define the convenient client-side methods **getValue** and **setValue**, which work only on the value property, and so do not require you to specify which property you wish to change.

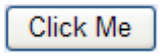
5.4 Buttons

Zen offers the following button-style controls:

- "<button>" — The user clicks a button that can trigger further actions
- "<image>" — The user clicks an image that can trigger further actions
- "<submit>" — The user clicks a button that submits a form

5.4.1 <button>

The <button> component is a simple wrapper for the HTML `<input type="button">` element. A Zen <button> looks like this:



<button> has the following attributes:

Attribute	Description
Control component attributes	<button> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." Control attributes include *onclick*, which determines what happens when a user clicks the button. If you want a user click to cause the form to be submitted, use the <submit> control instead of <button>.
caption	Text displayed on this button. The above example uses: `<button caption="Click Me"/>` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types." The *caption* value can be a literal string, or it can contain a Zen #()# runtime expression.

5.4.2 <image>

The <image> control displays a static image. <image> can be used simply to display an image, or it can serve as a button if you specify an *onclick* event for it. The <image> component offers several different properties whose values can specify the image to display:

- src can specify the URI of an image.
- streamId can specify the OID value for a binary stream object containing the image. You can work with the streamId property programmatically, but you cannot specify it as an XML attribute. See details following the table.
- value (a control component attribute) is used when the image is bound to a property within a data controller that contains binary stream data. In this case the value specifies an encrypted stream OID value for a binary stream object containing the image.

Images that are binary stream objects are served via the CSP stream server. The OID value is encrypted (using the current session key) when it is sent to the client. If this value is returned to the server, it is automatically decrypted. This makes it possible to place sensitive data on the client for future processing on the server without letting a user view this data.

The <image> element has the following attributes:

Attribute	Description
Control component attributes	<image> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." For the <image> element, *value* plays a special role, as described in this section. Also, *controlClass* does not apply to <image>, but *controlStyle* does apply.
alt	A string of text to display in place of the image if it is unavailable. This becomes the *alt* attribute value for the HTML <img> element on the displayed page. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
srcDisabled	If provided, *srcDisabled* is the URI of an image to display when the *disabled* attribute for this <image> is set to true.

Attribute	Description
srcMissing	If provided, *srcMissing* is the URI of an image to display when a *value* for this image is missing. *srcMissing* is used when this image is bound to a property of a data controller and there is no *value* for the bound property. There is a default for *srcMissing* if you do not provide it. The default is a small blank space image: `"images/spacer.gif"`
src	If provided, *src* is the URI of an image to display. If *src* is the relative pathname of a file, it is understood to be relative to the Caché installation directory. Typically this path identifies the images subdirectory for your Zen application, for example: `<image id="myFrame" src="/csp/myApp/images/myPic.png" />`
text	If defined, this string provides a text value to associate with this image. *text* is used as both the logical value and the display value when this image is the source of a drag and drop operation. To enable drag and drop features to work, the <page> that contains the <image> must have its *dragAndDrop* attribute set to true. For details, see the "Data Drag and Drop" section in this chapter.
value	The *value* attribute works differently for <image> than for other control components. First of all, <image> does not submit its *value* when the user submits the form. Secondly, the *value* plays an important role in serving the image. *value* is used when the image is bound to a property within a data controller that contains binary stream data. In this case, *value* specifies an encrypted stream OID value for a binary stream object containing the image.

The %ZEN.SVGComponent.image class offers a streamId property. If it has a value, the streamId is the OID value for a binary stream object containing the image.

There is no way to assign a streamId directly in the XData block, because you must first acquire an OID for the object you are referencing. You can only assign a value to the streamId property by working with the image component programmatically, in **%OnAfterCreatePage**.

To specify an image using a streamId, use this technique:

1. Provide the <image> element in XData, that is:

   ```
   <image id="img"/>
   ```

2. In the **%OnAfterCreatePage** method for the Zen page class that displays the image, provide code that references the <image> element and provides a value for its streamId, for example:

   ```
   Set img = ..%GetComponentById("img")
   Set stream=##class(%FileBinaryStream).%New()
   Set filename=$system.CSP.GetFileName("/csp/samples/ClassLogo.jpg")
   Do stream.LinkToFile(filename)
   Set oid=stream.%Oid()
   Set img.streamId = oid
   ```

Note: If both streamId and *src* values are provided for an image, streamId overrides *src*.

5.4.3 <submit>

<submit> is a special type of button that submits a form. For details, see the section "Processing a Form Submit" in this chapter.

<submit> has the following attributes:

Attribute	Description
Control component attributes	<submit> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." You do not supply an *onclick* attribute for a <submit> control, because the purpose of clicking an <submit> control is to submit the form on which it appears.
caption	Text displayed on this button. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types".
action	String giving the action code associated with this <submit> button. This value is passed along to the server-side **%OnSubmit** method of the page that contains the <submit>. If not provided, the default string is "submit".
nextPage	URI of the page to display after this form is successfully submitted. If a <submit> button defines a *nextPage* value, it overrides the *nextPage* value for the form that contains the <submit>.

5.5 Text

Zen offers the following text-style controls:

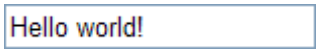
- "<label>" — Displays a text label
- "<text>" — The user inputs text
- "<textarea>" — The user inputs multiple lines of text
- "<password>" — The user inputs a text password

5.5.1 <label>

The <label> control passively displays a static text *value*. Zen submits the <label> along with other controls on the <form>. <label> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes."

5.5.2 <text>

The Zen <text> control is a wrapper around the HTML `<input type="text">` element. Zen <text> displays a text input box like this:

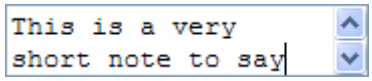


<text> has the following attributes:

Attribute	Description
Control component attributes	<text> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <text> *value* is a string of text.
autocomplete	Indicates whether the value of this text control can be automatically completed by default by the browser.
maxlength	Maximum number of characters that the user may enter within this text control.
placeholder	A placeholder that specifies a short hint describing the expected value of an input field, for example, a sample value or a short description of the expected format. The hint is displayed in the input field when it is empty.
size	Integer indicating the HTML width of the input area for this text control. The default *size* is 20.
spellcheck	If true, spellcheck is enabled. This is an HTML5 attribute. It works correctly only on HTML5 compliant browsers. Supported only by IE10 and higher. The default value is true. *spellcheck* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

5.5.3 <textarea>

The Zen <textarea> control is a wrapper around the HTML `<textarea>` element. Zen <textarea> displays a multi-line text input box, like this:



<textarea> has the following attributes:

Attribute	Description
Control component attributes	<textarea> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <textarea> *value* is a string of text that may or may not include line break characters, depending on what the user types. Keep in mind that many browsers do not cope well with long lines of unbroken text; that is, greater than 4K characters with no white space.
cols	Number of columns in the <textarea> control. The default is 19.
rows	Number of rows in the <textarea> control. The default is 2.
spellcheck	If true, spellcheck is enabled. This is an HTML5 attribute. It works correctly only on HTML5 compliant browsers. Supported only by IE10 and higher. The default value is true. *spellcheck* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

5.5.4 <password>

The Zen <password> control is a wrapper around the HTML `<input type="password">` element. Zen <password> displays a text input box for passwords. Any text that the user enters into the <password> control is echoed as a dot instead of being displayed on the screen. For example:

Note: For an example of a user login page that use the <password> element to control access to a Zen application, see the section "Controlling Access to Applications" in the "Zen Security" chapter of *Developing Zen Applications*.

<password> has the following attributes:

Attribute	Description
Control component attributes	<password> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <password> *value* is a string of text.
maxlength	Maximum number of characters that the user may enter within this control.
size	Integer indicating the HTML width of the input area for this control. The default *size* is 20.

5.6 Selections

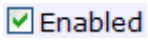
Zen offers the following selection controls:

- "<checkbox>" — The user selects or clears a check box
- "<multiSelectSet>" — The user selects or clears one or more check boxes.
- "<fileUpload>" — The user browses to choose a file
- "<colorPicker>" — The user selects one color from a palette
- "<radioSet>" — The user clicks one in a simple row of radio buttons
- "<radioButton>" — Radio buttons might be placed anywhere on the page

The difference between <radioButton> and <radioSet> is that <radioSet> is simpler to lay out. Use <radioSet> for a concise list of choices. Use <radioButton> when you want a more complex page layout that provides intervening information or images in between the radio button choices, or when you want to place radio buttons in a vertical group.

5.6.1 <checkbox>

The Zen <checkbox> control is a wrapper around the HTML `<input type="check">` element. The Zen <checkbox> control displays a caption next to the check box and detects user mouse clicks on the caption text as well as on the check box. Unlike an HTML check box, the Zen <checkbox> control always submits a *value*. Zen <checkbox> looks like this:

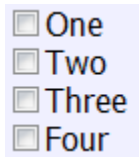


<checkbox> has the following attributes:

Attribute	Description
Control component attributes	<checkbox> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." <checkbox> always has its *clientType* set to "boolean". This means the <checkbox> *value* is expressed as 1 or 0 in XData Contents and server-side code, and true or false in client-side code. A *value* of 1 or true means the check box is currently selected; 0 or false means it is clear.
caption	Text displayed to the right of the check box. The above example uses: `<checkbox caption="Enabled"/>` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types." The *caption* value can be a literal string, or it can contain a Zen #()# runtime expression.
captionClass	Name of a CSS style class to apply to the *caption* text. The default is: `checkboxCaption`

5.6.2 <multiSelectSet>

The <multiSelectSet> control displays a column of check buttons to show a complete set of choices. The user can select one or more of the listed choices. A Zen <multiSelectSet> looks like this:



To define a <multiSelectSet>, provide a *valueList* for the user to choose from. If you also provide a corresponding *displayList* it provides the displayed captions for the user to see. For example, you could produce the sample <multiSelectSet> illustrated above by providing the following statement in XData Contents:

XML

```
<multiSelectSet displayList="One,Two,Three,Four"
    valueList="1,2,3,4" />
```

<multiSelectSet> has the following attributes:

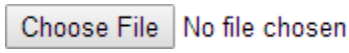
Attribute	Description
Control component attributes	<multiSelectSet> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <multiSelectSet> *value* is a comma-separated list of the values that are currently checked. Items appear in this list in the same order as in the *valueList* It is also possible to programmatically set the *value* of the <radioSet> to any arbitrary value using the client-side **setValue** method. If the *value* of the <multiSelectSet> does not correspond to any item in the *valueList*, all check boxes in the set appear clear.

Attribute	Description
captionClass	Name of the CSS style class to apply to captions for check boxes within this <multiSelectSet>. The default is the built-in CSS style class `multiSelectSetCaption`.
choiceColumn	Serves the same function as the *choiceColumn* attribute in dataCombo, see "<dataCombo> General Attributes."
displayList	Comma-separated list of choices to display for this <multiSelectSet>. *displayList* applies only if a *valueList* is defined. Display values may differ from the actual logical values. If there is an empty value ("") within the items in the *displayList*, as in: `valueList=",A,B,C"` Then an additional check box is displayed for the empty value. The caption for this empty value is specified by the *emptyCaption* attribute. The *displayList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *displayList* string must remain a comma-separated list.
emptyCaption	The default caption to use for any check boxes in this <multiSelectSet> that have an empty value ("") assigned to them in the *displayList*. If you do not specify an *emptyCaption*, the default caption is: `None` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
titleList	Comma-separated list of tooltip text strings for each check box in the <multiSelectSet>. The *titleList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *titleList* string must remain a comma-separated list.
valueColumn	Serves the same function as the *valueColumn* attribute in dataCombo, see "<dataCombo> General Attributes."

Attribute	Description
valueList	Comma-separated list of logical values for the <multiSelectSet>. If there is an empty value ("") in the *valueList* (for example, "A,,B,C"), then an additional button is displayed for the empty value. The label for the empty value is specified by the *emptyCaption*. The value of the <multiSelectSet> is a comma-separated list of the values that are currently checked. Items appear in this list in the same order as in the *valueList*. Use the *displayList* to provide a corresponding list of choices to display to the user. Zen assumes that every value within the set is distinct. If a *valueList* contains duplicate items, as in: `valueList="A,A,A"` Then user selections produce unexpected behavior.

5.6.3 <fileUpload>

The <fileUpload> component is a simple wrapper for the HTML `<input type="file">` element. A Zen <fileUpload> control looks like this:



The precise appearance and text used in the control is determined by the browser, and therefore varies with different browsers. When the user has selected a file, the text “No file chosen” is replaced with the filename, or text indicating the number of files selected in the case of multiple upload.

A user clicks the button to open a file open dialog and navigate to the file of interest, then select and open the file. The next action depends on the browser. When the user submits the form on which the <fileUpload> appears, on some browsers, the value of the <fileUpload> component is the full pathname including the filename. On many modern browsers, the value is the filename only. Browser manufacturers offer this denial of visibility into the full path as a security measure. If included in the browser design, this security measure cannot be overruled by a Zen application.

To be sure that your Zen application is independent of the user’s choice of browser, you can strip the full pathname from the returned value of the <fileUpload> component. As an example, suppose your <fileUpload> component has an *id* of getFile. Inside your **%OnSubmit** method, you can have these lines of ObjectScript code:

Class Member

```
ClassMethod %OnSubmit(pSubmit As %ZEN.Submit) As %Status
{
 Set file = pSubmit.%GetValue("getFile")
 Set file = ##class(%Library.File).GetFilename(file) // strip out the path on IE
 // ... other lines of code here ...
 Quit $$$OK
}
```

<fileUpload> has the following attributes:

Attribute	Description
Control component attributes	<fileUpload> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <fileUpload> *value* is a string that is the full pathname of a file.
accept	Comma-separated list of MIME types that can be uploaded. If provided, this value is used as the *accept* attribute value for the HTML <input> element.
maxlength	Maximum number of characters that the user may enter in this control. This property was used in the past, when browsers allowed the user to type a filename into this control. It is retained primarily for compatibility with earlier versions of Caché
multiple	If true, allow multiple files to be uploaded at once. The default value is false. This property requires an HTML5 compliant browser.
size	Integer indicating the HTML width of the input area for this control. The default *size* is 20.

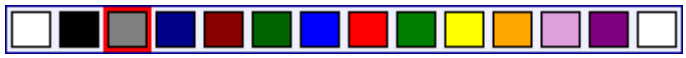
The fileSelect dialog offers a mechanism to select files on the server. See the section "File Selection Dialog Window."

Important: If you use the <fileUpload> component, you need to be aware of an important security restriction on the file upload control that is enforced by most browsers. The following paragraphs describe this restriction.

Most modern browsers regard the *value* field of a file upload control to be read-only for security purposes. The rationale behind this is that, if it were possible to set the value field via a script, it would be trivial to link a form submit event to any other event generator on the page. A devious programmer could, for example, embed a hidden form, either not displayed or clipped to an area only 1 pixel square, and program it to covertly copy files off the client machine every time the mouse pointer moved. For this reason, the value of the file to be uploaded on a form submit can only be set by direct user action on most browsers. No modern browsers allow the value to be set quietly in the background.

5.6.4 <colorPicker>

The <colorPicker> component displays a row of color choices. The user can click on a color to select it. <colorPicker> offers a simple alternative to the complex palette in <colorPane>. A Zen <colorPicker> looks like this:



<colorPicker> has the following attributes:

Attribute	Description
Control component attributes	<colorPicker> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <colorPicker> *value* is a string that identifies the most recently selected CSS color value.
colorList	Comma-separated list of CSS color values to display within the control, from left to right. The default *colorList* value is as shown in the example above: `",black,gray,darkblue,darkred,darkgreen,blue,red,green,yellow,orange,plum,purple,white"`

5.6.5 <radioSet>

The <radioSet> control displays a row of radio buttons to show a complete set of choices. A Zen <radioSet> looks like this:

5.6.5.1 <radioSet> General Attributes

To define a <radioSet>, provide a *valueList* for the user to choose from and a corresponding *displayList* for the user to see. For example, you could produce the sample <radioSet> illustrated above by providing the following statement in XData Contents:

XML

```
<radioSet id="QuarkStatus" name="QuarkStatus"
          displayList="Up,Down,Charmed,Strange,Top,Bottom"
          valueList="U,D,C,S,T,B"/>
```

<radioSet> has the following general-purpose attributes.

Attribute	Description
Control component attributes	<radioSet> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <radioSet> *value* is the logical value of the currently selected button in the set. This logical value comes from the corresponding *valueList* entry for the <radioSet>. It is also possible to programmatically set the *value* of the <radioSet> to any arbitrary value using the client-side **setValue** method. If the *value* of the <radioSet> does not correspond to any item in the *valueList*, all buttons in the set appear clear.
Data source attributes	<radioSet> has similar attributes to <tablePane> for specifying the data source. <radioSet> supports *maxRows*, *queryClass*, *queryName*, and *sql*. See "<radioSet> Query Attributes."
captionClass	Name of the CSS style class to apply to captions for radio buttons within this <radioSet>. The default is the built-in CSS style class `radioSetCaption`.
choiceColumn	Serves the same function as the *choiceColumn* attribute in dataCombo, see "<dataCombo> General Attributes."
displayList	Comma-separated list of choices to display for this <radioSet>. *displayList* applies only if a *valueList* is defined. Display values may differ from the actual logical values. If there is an empty value ("") within the items in the *displayList*, as in: `valueList=",A,B,C"` Then an additional button is displayed for the empty value. The caption for this empty value is specified by the *emptyCaption* attribute. The *displayList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *displayList* string must remain a comma-separated list.

Attribute	Description
emptyCaption	The default caption to use for any buttons within this <radioSet> that have an empty value ("") assigned to them in the *displayList*. If you do not specify an *emptyCaption*, the default caption is: `None` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
titleList	Comma-separated list of tooltip text strings for each radio button in the <radioSet>. The *titleList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *titleList* string must remain a comma-separated list.
valueColumn	Serves the same function as the *valueColumn* attribute in dataCombo, see "<dataCombo> General Attributes."
valueList	Comma-separated list of logical values for the <radioSet>. One of these logical values becomes the <radioSet> *value* whenever the user clicks on the corresponding button. Use the *displayList* to provide the corresponding list of choices to display to the user. Zen assumes that every value within the set is distinct. If a *valueList* contains duplicate items, as in: `valueList="A,A,A"` Then user selections produce unexpected behavior.

5.6.5.2 <radioSet> Query Attributes

Rather than provide a *valueList* and a *displayList*, a <radioSet> can indicate a data source for its buttons via an SQL query. <radioSet> offers a number of attributes for this purpose. When you use this technique, the columns returned by the query determine what is displayed in the <radioSet> as follows:

- If the %ResultSet has one column, the contents of this column are used as both the logical and display values within the radioSet.
- If the %ResultSet has two (or more) columns, the contents of the first column supply the logical value and the contents of the second column supply the display values.

<radioSet> provides several attributes that support using a query to generate the result set. For details and examples, see the following sections in the chapter "Zen Tables":

- How to use query attributes with <radioSet>:
  - Data Sources (*maxRows*)
  - Specifying an SQL Query (*sql*)
  - Referencing a Class Query (*queryClass*, *queryName*)
- How to provide <parameter> elements within <radioSet>:
  - Query Parameters

5.6.6 <radioButton>

The Zen <radioButton> control is a wrapper around the HTML `<input type="radio">` element with some enhanced capabilities. The Zen <radioButton> control has the following attributes:

Attribute	Description
Control component attributes	<radioButton> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." *name* (not *id*) establishes the association between radio buttons. You create a set of associated radio buttons by adding <radioButton> elements to XData Contents and assigning the same *name* value to each <radioButton> that in the set. At runtime, *value* always contains the *optionValue* of the button in the set that is currently selected. As soon as the user selects one of the buttons in the set, Zen resets the *value* of every button in this set to the *optionValue* of the selected button. This makes it very easy for you to determine which button in the set is currently selected, by programmatically checking the value property of any button in the set.
caption	The caption text for this <radioButton>. Each button in the set needs a *caption* so that the user can distinguish between the choices. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types." The *caption* value can be a literal string, or it can contain a Zen #()# runtime expression.
captionClass	Name of the CSS style class to apply to the captions for this <radioButton>. The default is the built-in CSS style class `radioButtonCaption`.
optionValue	Defines a logical value to associate with this <radioButton>. The *optionValue* for each radio button in the set must be unique. The *optionValue* can be a literal string, or it can contain a Zen #()# runtime expression.

5.7 Lists

Zen offers the following list controls:

- "<select>" — You display a list box by using the Zen wrapper for HTML <select>
- "<listBox>" — You define a Zen list box with fixed options for the user to choose
- "<dataListBox>" — Zen generates a list box for you, based on a runtime query
- "<combobox>" — You define a Zen combo box with fixed options for the user to choose
- "<dataCombo>" — Zen generates a combo box for you, based on a runtime query
- "<lookup>" — You define a control that provides a way to select a value from a list of options.

A list box offers a simple list of options, but a combo box has two parts:

- A text control that displays the current value of the control

- A drop-down list that displays a set of options for the user to select

A combo box drop-down list may appear when activated by the user, for example when the user clicks the button beside the control. There are a number of ways to reveal the drop-down list. The user can click on an image or button, or the list can simply appear when the cursor has hovered over the control for some length of time. You can specify these details when you place a list control on a Zen form, or simply use the default characteristics that Zen provides.

5.7.1 <select>

The Zen <select> control is a wrapper around the HTML <select> element. Zen <select> produces a list from which the user can select an item.

To define a Zen <select> list, provide a *valueList* for the user to choose from and a corresponding *displayList* for the user to see. The following table describes these and other attributes for the Zen <select> component.

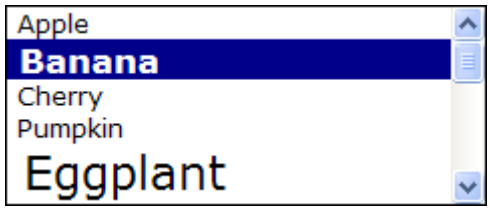
Attribute	Description

Attribute	Description
Control component attributes	<select> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The <select> *value* is a string indicating the user's current choice from the <select> list.
Data source attributes	Rather than provide a *valueList* and a *displayList*, a Zen <select> control can indicate a data source for its list via an SQL query. Zen <select> offers a number of attributes for this purpose. When you use this technique, the columns returned by the SQL query determine what is displayed within the <select> list as follows: • If the %ResultSet has one column, the contents of this column are used as both the logical and display values within the drop-down. • If the %ResultSet has two (or more) columns, the contents of the first column supply the logical value and the contents of the second column supply the display values. <select> supports the query attributes *maxRows*, *queryClass*, *queryName*, and *sql*. For details and examples, see the following sections in the chapter "Zen Tables": • How to use query attributes with <select>: – Data Sources (*maxRows*) – Specifying an SQL Query (*sql*) – Referencing a Class Query (*queryClass*, *queryName*) • How to provide <parameter> elements within <select>: – Query Parameters
choiceColumn	Serves the same function as the *choiceColumn* attribute in dataCombo, see "<dataCombo> General Attributes."
displayList	Comma-separated list of values to display for this <select> list. *displayList* applies only if a *valueList* is defined. Display values may differ from the actual logical values. The *displayList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *displayList* string must remain a comma-separated list.
emptyText	Text that is displayed in the "empty" item added if *showEmpty* is true. The default is "".
showEmpty	When true, the <select> outputs an extra at the top of its drop-down box. The attribute *emptyText* provides content for this item. The default value of *emptyText* is "". Typically, this is the desired behavior, so the default value of *showEmpty* is true. Regardless of the value of *showEmpty*, this blank row does not display when the <dataCombo> has its *required* attribute set to true. *showEmpty* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

Attribute	Description
size	Number of rows to display in the <select> list.
valueColumn	Serves the same function as the *valueColumn* attribute in dataCombo, see "<dataCombo> General Attributes."
valueList	Comma-separated list of logical values for the <select> list.

5.7.2 <listBox>

The Zen <listBox> control displays a list box, whose options may be individually formatted.



The Zen <listBox> control is *not* a wrapper around HTML <select>. The Zen <listBox> is implemented using HTML primitives. This allows the <listBox> to provide functionality not available with HTML <select>, including:

- Greater control over the contents of the list
- Solutions to problems with Internet Explorer interoperating with CSS

5.7.2.1 <listBox> Options

The simplest way to define a Zen list box is to provide a set of <option> elements inside a <listBox> component. The following statements produce the sample list box shown previously:

XML

```
<listBox id="listBox" label="listBox" listWidth="240px"
         onchange="zenPage.notifyOnChange(zenThis);"
         value="2">
  <option value="1" text="Apple" />
  <option value="2" text="Banana" style="font-size: 1.5em; "/>
  <option value="3" text="Cherry" />
  <option value="4" text="Pumpkin" />
  <option value="5" text="Eggplant" style="font-size: 2.1em; "/>
</listBox>
```

The <option> element is the XML projection of the %ZEN.Auxiliary.option class and supports the attributes described in the following table.

Attribute	Description
style	CSS style to apply to this option.
text	Display value. This is the text that the user sees in the list box. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
value	Logical value. This is the value that Zen submits for this control when it submits the form. You must always provide both *text* and *value* for each option in the list.

When you work with %ZEN.Component.listBox programmatically, you work with <option> elements as members of the options property, a list collection. Each <option> in the <listBox> becomes a member of the options collection, associated with an ordinal position: 1, 2, 3, etc.

5.7.2.2 <listBox> General Attributes

<listBox> and <dataListBox> have the following general-purpose attributes in common.

Table 5–2: List Box Component Attributes

Attribute	Description
Control component attributes	A list box has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The list box *value* is a string of text that represents the currently selected logical value of the control.
listHeight	CSS length value. If defined, *listHeight* overrides the default height of the list box window, which is 250px.
listWidth	CSS length value. If defined, *listWidth* overrides the default width of the list box window, which is 250px.
selectedIndex	0-based index of the currently selected option in the list box. The default *selectedIndex* is –1 (nothing is selected).
text	The display text that corresponds to the currently selected item. Do not access the *text* value directly; use `getProperty('text')` instead.

5.7.2.3 <listBox> Drag and Drop

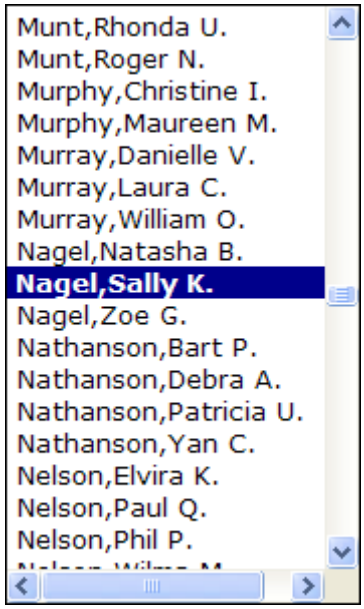
Like all controls, <listBox> supports data drag when the <listBox> *dragEnabled* attribute is true. This allows the application user to drag values from the list box and drop them on other controls. If *dropEnabled* is also true, values from other components can be dropped onto this <listBox>, where they are appended to the end of the list. This is different from the simpler replacement that drag and drop provides for other Zen controls. This special case for <listBox> enables the user to drag options from one list to another. If both drag and drop are enabled, the user can rearrange the order of list items within the same <listBox> by using the mouse to drag list items into a new position and drop them there.

To enable drag and drop features to work, the <page> that contains the <listBox> must have its *dragAndDrop* attribute set to true. For details, see the "Data Drag and Drop" section at the beginning of this chapter.

5.7.3 <dataListBox>

A <dataListBox> is a specialized type of <listBox> that presents the user with a list of options obtained at runtime via an SQL query. Unlike <listBox>, <dataListBox> does not allow you to specify <option> elements or to set styles for individual options in the list. This is because <dataListBox> uses a query to obtain a dynamic list of options. There are no statically defined options in a <dataListBox>, so there are no <option> elements.

A <dataListBox> with an item selected looks like this:



<dataListBox> has the following attributes:

Attribute	Description
List box attributes	<dataListBox> has the same general-purpose attributes as <listBox>. For descriptions, see the section "<listBox> General Attributes."

Attribute	Description
Data source attributes	The <dataListBox> provides its list by creating, executing, and fetching from a %ResultSet object on the server. You can specify how to create this %ResultSet object using the attributes that the <dataListBox> inherits from its parent class %ZEN.Component.querySource. The columns returned by the SQL query determine what is displayed within the <dataListBox> list, as follows: • If the %ResultSet has one column, the contents of this column are used as both the logical and display values within the drop-down. • If the %ResultSet has two (or more) columns, the contents of the first column supply the logical value and the contents of the second column supply the display values. For details and examples of using QuerySource attributes with <dataListBox>, see the following sections in the chapter "Zen Tables": • How to indicate a data source for a <dataListBox>: – Data Sources (*maxRows*) – Specifying an SQL Query (*sql*) – Generating an SQL Query (*groupByClause*, *orderByClause*, *tableName*, *whereClause*) – Referencing a Class Query (*queryClass*, *queryName*) – Using a Callback Method (*OnCreateResultSet*, *OnExecuteResultSet*) • How to provide <parameter> elements within <dataListBox>: – Query Parameters
sqlLookup	<dataListBox> has an *sqlLookup* attribute that works in a manner similar to <dataCombo>. The primary difference in behavior is that if the value found by *sqlLookup* is not already visible in the list, <dataListBox> does not bring it into view. For additional information, see the section "<dataCombo> Logical and Display Values."
OnDrawItem	Name of a server-side callback method in the Zen page class. Find further information following this table.
itemCount	(Read-only) Number of options within the list. This is calculated when the query for this component is run. It has no value until the list is displayed.

The *OnDrawItem* method returns the HTML to display within the cell for the given item. This is the place to escape special characters or make other last-minute adjustments to the HTML before displaying it.

Zen invokes this method when it first draws the list box, automatically passing it the following parameters:

- %ResultSet — the result set from *OnCreateResultSet*, if this <dataListBox> uses a result set to generate its content
- %String — the logical value for the item
- %String — the display value for the item

The callback must return a %String that contains the resulting HTML. The following example shows a valid method signature:

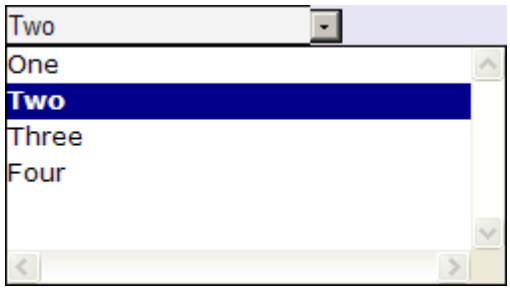
Class Member

```
Method DrawItem(pRS As %ResultSet,
                pValue As %String,
                pText As %String) As %String
{
 Set tx=pText
 Set tx=$REPLACE(tx,"&amp;eacute;",$ZCVT($CHAR(233),"O","HTML"))
 Set tx=$REPLACE(tx,"&amp;ntilde;",$ZCVT($CHAR(241),"O","HTML"))
 Quit tx
}
```

To use the above method as the callback, the developer would set `OnDrawItem="DrawItem"` for the <dataListBox>.

5.7.4 <combobox>

The Zen <combobox> provides a text field with a drop-down list below it:



Unlike some other controls described in this chapter, the Zen <combobox> control is *not* a wrapper around HTML <select>. The Zen <combobox> is implemented using HTML primitives. This allows the <combobox> to provide functionality not available with HTML <select>, including:

- The ability to edit values in the text box
- Greater control over the contents of the list
- Solutions to problems with Internet Explorer interoperating with CSS

5.7.4.1 <combobox> Logical and Display Values

To define a <combobox>, provide a *valueList* for the user to choose from and a corresponding *displayList* for the user to see. The following table describes these <combobox> attributes.

Attribute	Description
displayList	A comma-separated list of values to display for the list in this combo box. *displayList* applies only if a *valueList* is defined. Display values may differ from the actual logical values. The *displayList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *displayList* string must remain a comma-separated list.
valueList	*valueList* overrides any <option> elements provided within the <combobox> (see the discussion following this table). The *valueList* is a comma-separated list of logical values for the drop-down list in this <combobox>. If you provide a *valueList*, you must also provide a *displayList*.

5.7.4.2 <combobox> Options

Rather than provide a *valueList* and a *displayList*, you can define a <combobox> by providing a set of <option> elements inside a <combobox> component. <option> is more flexible than a *valueList* and *displayList* because it allows you to apply a CSS style to each of the list entries individually. For example:

XML

```
<combobox id="comboboxEdit" label="combobox Editable" editable="true">
  <option value="1" text="Apple" />
  <option value="2" text="Banana" style="font-size: 2.5em; "/>
</combobox>
```

The <listBox> control also uses <option> to add entries. See the section "<listBox Options" for more information.

5.7.4.3 <combobox> General Attributes

<combobox> and <dataCombo> have the following attributes in common.

Table 5–3: Combo Box Component Attributes

Attribute	Description
Control component attributes	A combo box has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." The combo box *value* is a string of text that represents the currently selected logical value of the control.
buttonCaption	Caption used for the button when the *comboType* is "button". Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
buttonImage	URI of the image to display for the combo button in its normal state.
buttonImageDown	URI of image to display for the combo button in its down (pressed) state.
buttonTitle	Popup title used for the drop-down button when *comboType* is "button" or "image". Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

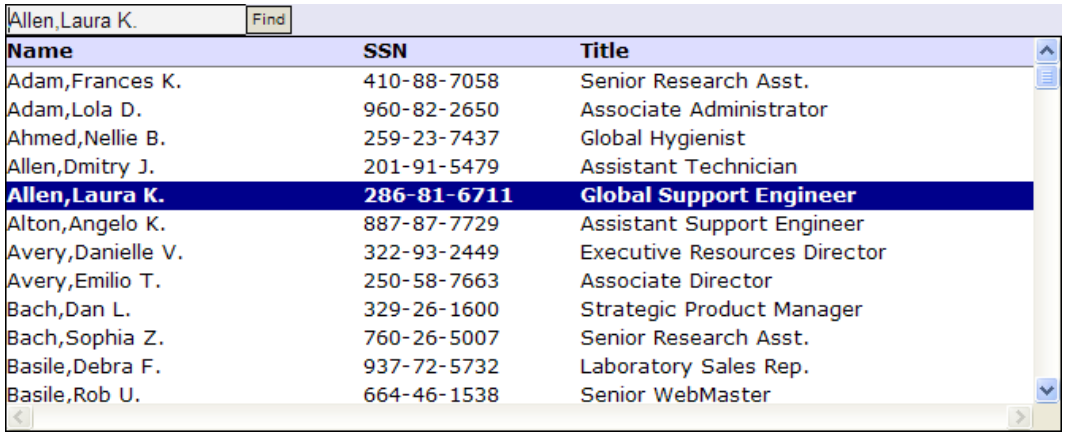
Attribute	Description
comboType	How the drop-down box is activated for the combobox: • "image" indicates that a user-clickable image should be displayed next to the combo box text box. This is the default. • "button" indicates that a button should be displayed next to the combo box text box. • "timer" indicates that the drop-down should appear shortly after the user enters a value within the combo box text box.
delay	When *comboType* is "timer", *delay* specifies how many milliseconds to wait after user finishes typing before showing the drop-down. The default is 250 milliseconds.
dropdownHeight	CSS length value. If defined, *dropdownHeight* overrides the default height of the drop-down window, which is 250px.
dropdownWidth	CSS length value. If defined, *dropdownWidth* overrides the default width of the drop-down window, which is 250px.
editable	If true, a user can directly edit the value within the input box as if it were a text field. The default is false. *editable* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
maxlength	Maximum number of characters that the user may enter within this control.
scrollIntoView	If true, Zen uses the JavaScript scrollIntoView function to try and make visible the currently selected item within the drop-down. *scrollIntoView* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
selectedIndex	0-based index of the currently selected option in the drop-down list. The default *selectedIndex* is –1 (nothing is selected).
size	HTML width of the text input area for this control. The default *size* is 20.
unrestricted	If true, and if *editable* is also true, values entered by the user may be used as the value of the control. If false, the value is restricted to one of the choices within the drop-down list. The default is false. *unrestricted* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

%ZEN.Component.combobox is a subclass of the %ZEN.Component.text control. This means you can use the various methods defined by the text control to manipulate the text box portion of a <combobox> or <dataCombo>.

5.7.5 <dataCombo>

A <dataCombo> is a specialized type of <combobox> that presents the user with a list of options obtained at runtime via an SQL query. Unlike <combobox>, <dataCombo> does not allow you to specify <option> elements or to set styles for individual options in the list. This is because <dataCombo> uses a query to obtain a dynamic list of options. There are no statically defined options in a <dataCombo>, so there are no <option> elements.

A <dataCombo> with text entered for a search looks like this:



5.7.5.1 <dataCombo> Query Attributes

The <dataCombo> provides its drop-down list by creating, executing, and fetching from a %ResultSet object on the server. Initially, the contents of the drop-down are empty, until a user action causes the drop-down to appear. At this point a call to the server fetches the drop-down contents. You can change this behavior by setting the *cached* attribute of the <dataCombo>.

You can specify how the <dataCombo> creates its %ResultSet object using the attributes that the <dataCombo> inherits from its parent class %ZEN.Component.querySource. The columns returned by the SQL query determine what is displayed within the <dataCombo> list, as follows:

- If the %ResultSet has one column, the contents of this column are used as both the logical and display values within the drop-down list.
- If the %ResultSet has two (or more) columns, the contents of the first column supply the logical value and the contents of the second column supply the display values. You can change which columns are used to provide the logical and display values using the *valueColumn* and *choiceColumn* attributes.
- If the %ResultSet has more than two columns, you can use the *displayColumns* and *columnHeaders* attributes to specify that the drop-down should display multiple columns.

For details and examples, see the following sections in the chapter "Zen Tables":

- "Data Sources" (*maxRows*)
- "Specifying an SQL Query" (*sql*)
- "Generating an SQL Query" (*groupByClause*, *orderByClause*, *tableName*, *whereClause*)
- "Referencing a Class Query" (*queryClass*, *queryName*)
- "Using a Callback Method" (*OnCreateResultSet*, *OnExecuteResultSet*)

5.7.5.2 <dataCombo> Query Parameters

The query used to provide the contents of the <dataCombo> drop-down list may contain one or more runtime ? parameters, such as:

```
WHERE Name %STARTSWITH ?
```

Values for query parameters can be provided in one of the following ways:

- The <dataCombo> can define a <parameter> list, as described in the "Zen Tables" topic "Query Parameters." The parameter values replace ? parameters in the order in which they appear in the SQL query. It is possible to modify the

values of these parameters programmatically from the client; if you do so, be sure to call the %ZEN.Component.dataCombo method **clearCache** so that the drop-down query is re-executed with the new values.

- If non-zero, *searchKeyLen* is the maximum number of search characters for Zen to take from the combo input box and pass as the first parameter to the SQL query that provides the contents of the drop-down list.

  If zero, the contents of the input box are not used as a query parameter. The default is 0.

  If *searchKeyLen* is non-zero, and *editable* is true, then the first *searchKeyLen* characters in the current contents of the input box are used as the value for the first query parameter (that is, `parms(1)`). The first member of the parameters list becomes the value of `parms(2)`, the second member of the parameter list becomes the value of `parms(3)`, and so on.

  If any parameter value is equal to "?" the current search key value (the value used for the first parameter) is used for this query parameter as well.

5.7.5.3 <dataCombo> Logical and Display Values

Any list box or combo box has two current values:

- Logical value — its actual stored *value* as returned by the **getValue** method
- Display value — the value that the user views and selects in the drop-down list

These two values are usually different, but they can be the same.

The Zen controls <select>, <listBox>, and <combobox> provide a fixed list of logical and display values. When an application sets the *value* of one of these controls, it is simple for the control to identify which display value is associated with the new logical value.

<dataListBox> and <dataCombo> acquire their values dynamically, so additional steps are needed to match logical and display values. When an application sets the local *value* of a <dataCombo> control, internally the <dataCombo> tries to find the display value that best matches this logical value. This works differently on the server and client:

- On the server, <dataCombo> executes the SQL statement defined by its *sqlLookup* attribute.
- On the client, the <dataCombo> first looks for a match for a given logical value within its drop-down cache. If it does not find a match, it calls a server method to execute the *sqlLookup* query.

For example, suppose you want to define a <dataCombo> to show a set of Customer names; the display value is Name while the logical value is the ID of the Customer. To do this you define a <dataCombo> with two SQL statements, as follows:

XML

```
<dataCombo id="MyCombo"
 sql="SELECT ID,Name FROM MyApp.Customer WHERE Name %STARTSWITH ? ORDER BY Name"
 sqlLookup="SELECT Name FROM MyApp.Customer WHERE ID = ?"
 editable="true"
 searchKeyLen="10"
 />
```

This sample <dataCombo> definition has the following effects:

1. The SQL query provided by the *sql* attribute is invoked whenever the Zen page displays the drop-down list. It provides a set of logical and display values for the <dataCombo>. The <dataCombo> stores the results of the last *sql* query in a local cache.
2. The SQL query provided by the *sqlLookup* attribute is invoked to find a specific display value for a specific logical value. The *sqlLookup* ? parameter gets its value from the current logical value of the <dataCombo> control.
3. The *sqlLookup* query is also executed when the application tries to set the logical value of this <dataCombo> at runtime and the logical and display values are not already stored in the cache.

4. The *searchKeyLen* value in the example indicates that Zen passes up to the first 10 characters from the combo input box to the SQL query that provides the contents of the drop-down list.

The *sql* and *sqlLookup* queries can use query parameters as described in the section "<dataCombo> Query Parameters."

The *sqlLookup* attribute has the underlying data type %ZEN.Datatype.sql. This means it cannot be accessed from the client. Its value is encrypted (using the current session key) when it is sent to the client. If this value is returned to the server, it is automatically decrypted. This makes it possible to place sensitive data on the client for future processing on the server, without letting a user view this data.

If you use the *sqlLookup* attribute to retrieve a list of user choices from a table, and you also set the attribute *unrestricted* to true so that the user can enter values that do not exist in the underlying table, the lookup that you specify in *sqlLookup* does not occur. *unrestricted* has the effect of disabling *sqlLookup*.

The *sqlLookup* attribute value must escape any XML special characters. For example, in place of the less-than symbol < you must substitute the XML entity `<` as follows:

```
sqlLookup=
"select * from infonet_daten.abopos where lieferadresse=? and status&lt;9"
```

The following table lists XML special characters that cause problems when they appear in *sqlLookup* strings, and the XML entities to substitute for them.

Table 5–4: XML Entities for Use in sqlLookup Attribute Values

Character	XML Entity	Description
>	>	Right angle bracket or "greater than" symbol.
<	<	Left angle bracket or "less than" symbol.
&	&	Ampersand.
'	'	Single quotation mark or apostrophe. A string enclosed in single quotes needs the ' entity to represent the ' character.
"	"	Double quotation mark. A string enclosed in double quotes needs the " entity to represent the " character.

5.7.5.4 <dataCombo> General Attributes

In addition to the attributes described in previous sections, <dataCombo> supports the following general-purpose attributes.

Attribute	Description
Combo box attributes	<dataCombo> has the same general-purpose attributes as <combobox>. For descriptions, see the section "<combobox> General Attributes." There can be no statically defined options for a <dataCombo>, so it does not support the *displayList* or *valueList* attributes.
auxColumn	If there are multiple data columns displayed in the drop-down list, *auxColumn* is the 1-based column number of the column that provides an additional auxiliary value for this control. *auxColumn* provides a way to supply an additional value that is not the display or logical value. If the *auxColumn* value is not a valid column number, no auxiliary data is provided. The default *auxColumn* value is 0.

Attribute	Description
cached	If *cached* is true, when the page is first displayed, it executes a query to fetch the initial contents of the drop-down list, and sets the itemCount property to the number of items within the drop-down. The client uses these cached results instead of going back to the server to fetch the contents of the drop-down list. You can clear the drop-down cache at any time by modifying the search parameters for the query, or by invoking the **clearCache** method. The default value for *cached* is false. In this case the user must take action to request the drop-down list before the query fetches its contents. The *cached* attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
choiceColumn	If there are multiple data columns displayed within the drop-down list, *choiceColumn* is the 1-based column number of the column that provides the display value for this control. The default *choiceColumn* value is 2. If the supplied *choiceColumn* value is greater than the number of columns in the query, the second column is used.
clearOnLoad	If true, and this <dataCombo> is bound to a data controller, then the contents of the drop-down list are cleared whenever a new instance is loaded into the controller. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
columnHeaders	If defined, *columnHeaders* is a comma-separated list of column headers to display in the drop-down list. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types." Any localized *columnHeaders* string must remain a comma-separated list.
contentType	Indicates how display values should be rendered. Possible values are: • "text" — The display values are HTML-escaped before being rendered. This is the default. • "html" — The display values are not HTML-escaped before being rendered.
displayColumns	If there are multiple data columns in the %ResultSet for the <dataCombo>, *displayColumns* can provide a comma-separated list of 1–based column numbers. This list identifies which columns out of the %ResultSet should be displayed.
emptyText	Text that is displayed in the "empty" item added if *showEmpty* is true. The default is "".

Attribute	Description
itemCount	(Read-only) Number of items in the drop-down list. This value is set as a side effect of populating the drop-down list. It has no value until the list is displayed. If *cached* is true, when the page is first displayed, it executes a query to fetch the initial contents of the drop-down list, and sets the itemCount property to the number of items within the drop-down immediately after the page's **%OnAfterCreatePage** callback method is invoked.. The client uses these cached results instead of going back to the server to fetch the contents of the drop-down list. You can clear the drop-down cache at any time by modifying the search parameters for the query, or by invoking the **clearCache** method. The default value for *cached* is false. In this case the user must take action to request the drop-down list before the query fetches its contents.
loadingMessage	This message is temporarily displayed while a server-side query is running to populate the <dataCombo> list. The default is: `$$$Text("Loading...","%ZEN");` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
multiColumn	If true, and if the result set contains more than 2 columns, display multiple columns in the drop-down box. The default is true. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
onshowDropdown	The *onshowDropdown* event handler for the <dataCombo>. Zen invokes this handler just before the drop-down list is displayed. See "Zen Component Event Handlers." If the expression returns a value, this value is used as the filter value for the drop-down query instead of the value typed into the input box.
searchKeyLen	If non-zero, *searchKeyLen* is the maximum number of search characters for Zen to take from the combo input box and pass as the first parameter to the SQL query that provides the contents of the drop-down list. If zero, the contents of the input box are not used as a query parameter. The default is 0. If *searchKeyLen* is non-zero, and *editable* is true, then the first *searchKeyLen* characters in the current contents of the input box are used as the value for the first query parameter (that is, `parms(1)`). The first member of the parameters list becomes the value of `parms(2)`, the second member of the parameter list becomes the value of `parms(3)`, and so on. If any parameter value is equal to "?" the current search key value (the value used for the first parameter) is used for this query parameter as well.

Attribute	Description
showEmpty	When true, the <dataCombo> outputs an extra at the top of its drop-down box. The attribute *emptyText* provides content for this item. The default value of *emptyText* is "". Typically, this is the desired behavior, so the default value of *showEmpty* is true. Regardless of the value of *showEmpty*, this blank row does not display when the <dataCombo> has its *required* attribute set to true. *showEmpty* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
valueColumn	If there are multiple data columns in the %ResultSet for the <dataCombo>, *valueColumn* identifies the 1-based column number of the column that provides the logical value for this control. The default *valueColumn* is 1. If the supplied *valueColumn* value is greater than the number of columns in the query, the first column is used.

5.7.5.5 <dataCombo> Display Sequence

If you are interested in details of how Zen displays the <dataCombo>, the following table lists the internal events in sequence, organized by user viewpoint, browser-based execution, and server-side execution. Most of the communication details are handled by the CSP technology underlying Zen. For background information, see the "Zen Client and Server" chapter in *Using Zen*.

Table 5–5: <dataCombo> Display Sequence

	User Viewpoint	In the Browser	On the Server
1	Select the drop-down		
2		Drop-down is activated	
3		Send hyperevent to the server to execute SQL	
4			Receive hyperevent to execute SQL
5			Send hyperevent response (consisting of JavaScript code) to populate the drop-down unit
6		Server fills the drop-down list	
7	List fills with items		
8	User selects an item		
9		Drop-down item is selected	
10		Call client-side select event handler	
11		Send hyperevent to the server to start a ZenMethod	
12			Receive hyperevent to start ZenMethod
13			Update server-side DOM

	User Viewpoint	In the Browser	On the Server
14			Send hyperevent response (consisting of JavaScript code) to synchronize client DOM with server DOM
15		Update client-side DOM to reflect changes made on the server during the ZenMethod	
16		Update visual presentation of DOM-bound components to match new client-side DOM	
17	Form fields get filled in based on the selection		

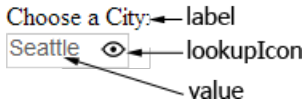
5.7.6 <lookup>

A <lookup> control is a specialized type of control that provides a way to select a value from a list of options. This is an HTML5 component. It works correctly only on HTML5 compliant browsers. This control supports the following attributes:

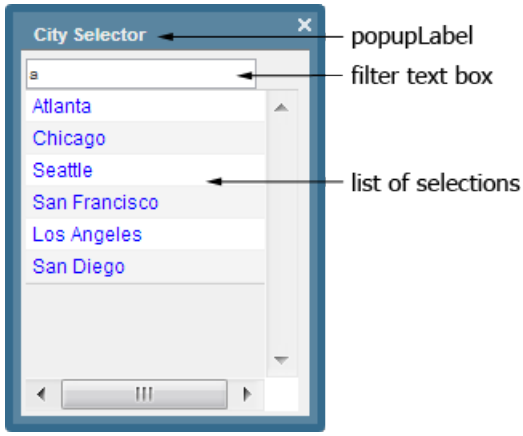
Attribute	Description
context	A context string used to determine the selection list for this component. The context string should take the form: ?parm1=value.
displayBinding	If this control is used in a form that is associated with a <dataController>, this attribute specifies the name of the property in the <dataController> that provides the display value for this control.
idProperty	The name of the property in data element that supplies the value of the control after the user has made a selection.
imageProperty	The name of the property in the data element that supplies a path to an image file. If this attribute is defined, and the specified property exists, the control shows the image in the popup instead of the text value. The image path is resolved relative to `CSP/broker` in the Caché installation directory. When the user makes a selection, the data element specified by *textProperty* supplies the value.
lookupIcon	The path to an image file that supplies the image used to invoke the lookup popup. The image path is resolved relative to `CSP/broker` in the Caché installation directory.
noResultsMessage	A message to show in the lookup popup when no selections are available. The default value is "Nothing to show!" Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
ongetdata	The ongetdata event handler, which returns a JavaScript array of data to display in the lookup popup. This event handler can return any array of object or literal values.
onshowPopup	The *onshowPopup* event handler, which is fired just before the popup is displayed.
popupLabel	The title to display in the popup. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

Attribute	Description
propertyList	A comma-separated list of property names used to create the display text in the popup list.
showFilter	Specify if there should be a filter text box in the pop up.
size	Size of the base (non popup) portion of this control. The number provided is multiplied by 10 to get the width.
styleList	A comma-separated list of CSS styles to apply to all the cells in the popup list.
text	Display value for this control. The *value* attribute contains the logical value.
textProperty	The name of the property in the data element that supplies the text value.

The following image shows the <lookup> control and identifies its main components.



This image shows the selection popup and identifies its main components. This example uses the *styleList* property to apply blue color to the selections. The property *showFilter* controls the presence of the text filter box. If the *ongetdata* event handler does not return any selections, the text provided by *noResultsMessage* replaces the list of selections .



The following code examples generate the <lookup> control illustrated in the previous images.

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">
 <lookup
  popupLabel="City Selector"
  label="Choose a City:"
  lookupIcon="MyApp/eye_16.png"
  ongetdata="return zenPage.myData();"
  styleList="color:blue,"
  idProperty="id"
  textProperty="text"
  value="Seattle"
 />
</page>
```

Class Member

```
ClientMethod myData() [ Language = javascript ]
{
 var data = [
  {id:1, text:'Boston',},
  {id:2, text:'New York'},
  {id:3, text:'Atlanta'},
  {id:4, text:'Chicago'},
  {id:5, text:'Tucson'},
  {id:6, text:'Seattle'},
  {id:7, text:'San Francisco'},
  {id:8, text:'Los Angeles'},
  {id:9, text:'San Diego'}
 ];
 return data;
}
```

The attribute *imageProperty* lets you use images in the selection popup. For example, the following <lookup> control and *ongetdata* callback:

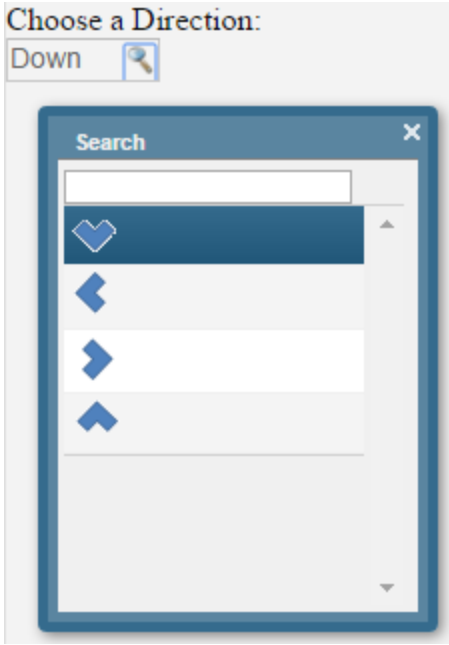
XML

```
<page xmlns="http://www.intersystems.com/zen" title="">
 <lookup
  label="Choose a Direction:"
  ongetdata="return zenPage.myData();"
  imageProperty="image"
  textProperty="text"
  />
</page>
```

Class Member

```
ClientMethod myData() [ Language = javascript ]
{
 var data = [
  {id:1, text:'Down', image:'images/arrowBD.png'},
  {id:2, text:'Left', image:'images/arrowBL.png'},
  {id:3, text:'Right', image:'images/arrowBR.png'},
  {id:4, text:'Up', image:'images/arrowBU.png'} ];
 return data;
}
```

Result in a popup that uses images, as shown in the following illustration:



You can use the attribute *propertyList* to construct the display text in the popup from more than one data element. This attribute supplies a comma-separated list of property names. The <lookup> control uses values associated with those names to populate the lookup list.

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">
 <lookup
  ongetdata="return zenPage.myData();"
  idProperty="lname"
  propertyList="fname,lname,dob"
  />
</page>
```

Results in a popup where each line contains the values from the fields `fname`, `lname`, and `dob` in the data. Only the field specified by *idProperty* becomes the value of the control after the user has made a selection.

You can also use CSS to control the appearance of items in the popup. The CSS class for lookup items is `lookupItem`. For example, the following CSS sets the background of each item in the popup to yellow.

```
.lookupItem {
  background-color:yellow;
}
```

5.8 Dates

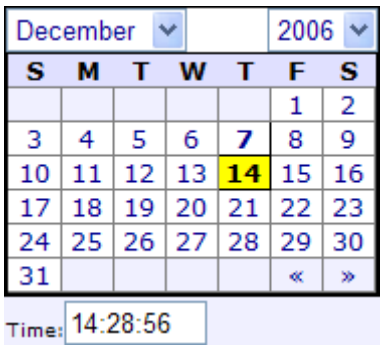
Zen offers the following date selection controls:

- "<calendar>" — The user selects dates from a popup calendar
- "<dateSelect>" — The user selects a month, a day, and a year
- "<dateText>" — The user can enter text or select a date

The components described in this topic are simple date selection controls that enable users to enter dates as values in forms. Zen offers another component, called <schedulePane>, which is not a date selection control. <schedulePane> displays a daily, weekly, or monthly calendar with time slots for each date. Users can define appointments and place them in the appropriate time slots. For details, see the "Schedule Calendar" section in the chapter "Other Zen Components."

5.8.1 <calendar>

The <calendar> component displays a navigable calendar, one month at a time. The user can view and select dates from this calendar. A <calendar> initially displays with the current date highlighted in bold (7 in the example below) and the currently selected date in bold with a yellow background color (14 in the example below). A Zen <calendar> looks like this:



<calendar> has the following attributes:

Attribute	Description
Control component attributes	<calendar> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." On the client, the <calendar> *value* is a string in the format YYYY-MM-DD. On the server, the <calendar> *value* has the %Timestamp datatype.
dayList	Comma-separated list of day abbreviations to show at the top of the calendar. If you do not provide a *dayList* value, the default is: `$$$Text("S,M,T,W,T,F,S")` The *dayList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *dayList* string must remain a comma-separated list.
defaultTime	Provides a default value for the time portion of the <calendar> *value*. The *defaultTime* is used as the initial time displayed in the popup calendar if the following conditions are met: • The property *showTime* is true. • The value supplied does not include a time portion. • *defaultTime* is a valid time string. The calendar itself defaults to a time value of "01:00:00" when a time is not specified or when the date value passed is invalid.
endYear	Four-digit end year number for the year selector in the calendar. If not defined, the default is the year portion of *maxDate*, if defined. Otherwise, the default *endYear* is 30 years from now.
firstDayOfWeek	Number (Sunday=0, Saturday=6) that specifies which day of the week is displayed as the starting day of the week. The default is 0 (Sunday).
fixedMonth	If true, this calendar displays a single month and provides no way for the user to change month and year. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
gapWidth	HTML length value giving the size of the gap between the month and year indicators at the top of the calendar. Setting this provides a way to adjust the width of the calendar. The above example uses the default of `40px`.
maxDate	String in the format YYYY-MM-DD (datatype %Timestamp on the server). If specified, this is the latest date the <calendar> accepts as its *value*. The *maxDate* value does not affect which years are displayed by the calendar unless *endYear* is omitted. The value supplied for *maxDate* can be a literal string, or it can contain a Zen #()# runtime expression.

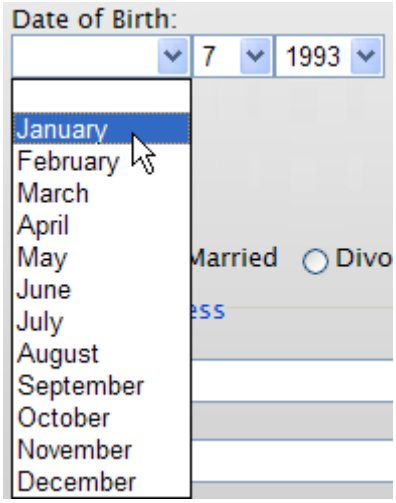
Attribute	Description
minDate	String in the format YYYY-MM-DD (datatype %Timestamp on the server). If specified, this is the earliest date the <calendar> accepts as its *value*. The *minDate* value does not affect which years are displayed by the calendar unless *startYear* is omitted. The value supplied for *minDate* can be a literal string, or it can contain a Zen #()# runtime expression.
month	Number (January=1, December=12) that specifies which month of the year is currently displayed by the calendar. This is not the same as the current <calendar> *value*, which includes a day and year, which can include time, and whose month may be different.
monthList	Comma-separated list of month names to display in the list of months that the user can choose from the calendar. If you do not provide a *monthList* value, the default is: `$$$Text("January,February,March,April,May,June,July,August,September,October,November,December")` *monthList* has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *monthList* string must remain a comma-separated list.
showTime	If true, this component displays a text entry field below the main calendar. In the example above, *showTime* is true. The default is false. The user can enter a time of day in the *showTime* field using the 24–hour time format HH:MM:SS. When the form is submitted, the <calendar> *value* accepts both date and time values in the following ODBC/JDBC timestamp format: `YYYY-MM-DD HH:MM:SS` *showTime* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
startYear	Four-digit start year number for the year selector in the calendar. If not defined, the default is the year portion of *minDate*, if defined. Otherwise, the default *startYear* is 10 years ago.
timeCaption	Caption text for the *showTime* field. If you do not provide a *timeCaption* value, the default is: `$$$Text("Time:")` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
year	Four-digit number that specifies which year is currently displayed by the calendar. This is not the same as the current <calendar> *value*, which includes a day and month, which can include time, and whose year may be different.

5.8.2 <dateSelect>

The <dateSelect> control displays three drop-down lists. From left to right, the lists allow the user to select a month, a day of the month, and a year. If the user types a character in any of these fields, Zen displays the closest matching value for

that list, if one exists, such as “March” or “May” for M. <dateSelect> is useful for cases like birth dates or expiration dates when a popup calendar like <dateText> can be cumbersome.

A Zen <dateSelect> component looks like this. In the example, the user is selecting from the month drop-down list.



<dateSelect> has the following attributes:

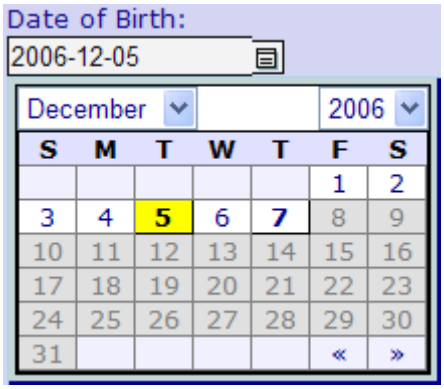
Attribute	Description
Control component attributes	<dateSelect> has the same general-purpose attributes as any Zen control. For descriptions, see the section “Control Attributes.” On the client, the <dateSelect> *value* is a string in the format YYYY-MM-DD. On the server, the <dateText> *value* has the %Timestamp datatype. Any HTML events defined for <dateSelect> (*onfocus*, *onclick*, etc.) apply to *each* of the three drop-down lists independently.
format	String that specifies the order in which to display the drop-down lists, from left to right. The default is `"MDY"`. Possible values are: • `"MDY"` — Month, Day, Year • `"DMY"` — Day, Month, Year • `"YMD"` — Year, Month, Day • `"YDM"` — Year, Day, Month • `"DM"` — Day, Month • `"MD"` — Month, Day • `"YM"` — Year, Month • `"Y"` — Year • `"M"` — Month The value supplied for *format* can be a literal string, or it can contain a Zen #()# runtime expression.

Attribute	Description
maxYear	Integer specifying the last year allowed in the year drop-down list. The default is the current year plus 20. The value supplied for *maxYear* can be a literal string, or it can contain a Zen #()# runtime expression.
minYear	Integer specifying the earliest year allowed in the year drop-down list. The default is 1900. The value supplied for *minYear* can be a literal string, or it can contain a Zen #()# runtime expression.
monthList	Comma-separated list of month names to display in the list of months that the user can choose from the calendar. If you do not provide a *monthList* value, the default is: `$$$Text("January,February,March,April,May,June,July,August,September,October,November,December")` *monthList* has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *monthList* string must remain a comma-separated list.
shortMonth	If true, this component shows the first 3 characters of the month names in the month drop-down list. In the example above, *shortMonth* is false. The default is false. *shortMonth* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types." *shortMonth* has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize the 3–character version of the month name into other languages.
showMonthNumber	If true, this component shows the ordinal month number along with month names in the month drop-down list. In the example above, *showMonthNumber* is false. The default is false. *showMonthNumber* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types." *showMonthNumber* has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize the longer text for the month selection into other languages.

5.8.3 <dateText>

The <dateText> control is essentially a combo box. It displays a text box as well as a button that, when pressed, displays a popup calendar. When the user enters a value into the text area of this control, Zen either converts this value into the closest matching date value, or it displays an invalid date message. As a comparison, see the <dateSelect> component.

A Zen <dateText> component looks like this:



<dateText> has the following attributes:

Attribute	Description
Control component attributes	<dateText> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." On the client, the <dateText> *value* is a string in the format YYYY-MM-DD. On the server, the <dateText> *value* has the %Timestamp datatype. However, you can change what the user sees in the <dateText> display; see the *format* and *separator* attributes.
dayList	Comma-separated list of day abbreviations to show at the top of the calendar. If you do not provide a *dayList* value, the default is: `$$$Text("S,M,T,W,T,F,S")` The *dayList* attribute has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *dayList* string must remain a comma-separated list.
defaultTime	Provides a default value for the time portion of the <dateText> control *value*. The *defaultTime* is used as the initial time displayed in the popup calendar if the following conditions are met: • The property *showTime* is true. • The *value* supplied does not include a time portion. • *defaultTime* is a valid time string. The calendar itself defaults to a time value of "01:00:00" when a time is not specified or when the date value passed is invalid.
firstDayOfWeek	Number (Sunday=0, Saturday=6) that specifies which day of the week is displayed as the starting day of the week. The default is 0 (Sunday).

Attribute	Description
format	String that specifies the display format for this component. The internal value of this control is always YYYY-MM-DD, but you can change what the user sees. The default is `"YMD"`. Possible values are: • `"YDM"` — Year, Day, Month • `"MDY"` — Month, Day, Year • `"DMY"` — Day, Month, Year The value supplied for *format* can be a literal string, or it can contain a Zen #()# runtime expression.
invalidDateMessage	Message displayed by control when the date entered by the user fails validation. The default is: `$$$Text("Invalid Date","%ZEN");` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
maxDate	String in the format YYYY-MM-DD (datatype %Timestamp on the server). If specified, this is the latest date the <dateText> accepts as its *value*. The *maxDate* value does not affect which years are displayed by the calendar unless *endYear* is omitted. The value supplied for *maxDate* can be a literal string, or it can contain a Zen #()# runtime expression.
minDate	String in the format YYYY-MM-DD (datatype %Timestamp on the server). If specified, this is the earliest date the <dateText> accepts as its *value*. The *minDate* value does not affect which years are displayed by the calendar unless *startYear* is omitted. The value supplied for *minDate* can be a literal string, or it can contain a Zen #()# runtime expression.
monthList	Comma-separated list of month names to display in the list of months that the user can choose from the calendar. If you do not provide a *monthList* value, the default is: `$$$Text("January,February,March,April,May,June,July,August,September,October,November,December")` *monthList* has its ZENLOCALIZE datatype parameter set to 1 (true). This makes it easy to localize its text into other languages, and permits use of the $$$Text macros when you assign values to this property from client-side or server-side code. Any localized *monthList* string must remain a comma-separated list.
separator	Separator character to use between date segments. The default is the hyphen character (-). if the user enters the forward slash (/) instead of a hyphen, the <dateText> component accepts it, but the hyphen is the expected format. As an alternative to the default hyphen, you can use the *separator* attribute to specify some other character as the date separator. If time is also displayed, the time portion of the date is always separated from the data portion using the colon (:).

Attribute	Description
showTime	If true, this component displays a text entry field below the main calendar. In the example above, *showTime* is true. The default is false. The user can enter a time of day in the *showTime* field using the 24–hour time format HH:MM:SS. When the form is submitted, the <calendar> *value* accepts both date and time values in the following ODBC/JDBC timestamp format: `YYYY-MM-DD HH:MM:SS` *showTime* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
size	HTML width of the text input area for this control. The default is 15.
timeCaption	Caption text for the *showTime* field. If you do not provide a *timeCaption* value, the default is: `$$$Text("Time:")` Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
value	You can access the value property of a dateText control programmatically. When you do this, it bypasses the date matching function that Zen provides when the user sets the date by typing text in the input field.

5.9 Grid

5.9.1 <dynaGrid>

The <dynaGrid> control displays a two-dimensional, editable grid, similar to a spreadsheet. When the user clicks in a cell, an edit control appears in the cell and the user can edit the cell contents. The user presses **Enter** to save changes. A <dynaGrid> with a newly edited cell looks like this:

Expenses	Mon	Tue	Wed	Thu	Fri	Total
Breakfast	10.00				12.00	**22.00**
Lunch	4.50			65.34		**69.84**
Dinner	8.00		108.77			**116.77**
Other	12.95					**12.95**
Total	**35.45**		**108.77**	**65.34**	**12.00**	**221.56**

You can create a <dynaGrid> in one of the following ways:

- Specify a data set, rows, and columns. See the section "<dynaGrid> Data Set."
- Associate the <dynaGrid> with a data controller. See the chapter "Model View Controller."

5.9.1.1 <dynaGrid> Data Set

The data displayed within the <dynaGrid> is supplied by a %ZEN.Auxiliary.dataSet object. dataSet is a special data container object that is used to define one-, two-, or three-dimensional data in a form that can be easily transported between the server and client.

When a <dynaGrid> object is first created, it automatically creates a two–dimensional dataSet object with the number of rows (dimension 1) and columns (dimension 2) specified by the number of <gridRow> and <gridColumn> entries within the <dynaGrid> definition in XData Contents. The following example specifies four rows and four columns for the initial dataSet object:

XML

```
<dynaGrid id="myGrid" OnCreateDataSet="CreateDS">
    <gridColumn width="100"/>
    <gridColumn width="100"/>
    <gridColumn width="100"/>
    <gridColumn width="100"/>
    <gridRow />
    <gridRow />
    <gridRow />
    <gridRow readOnly="true" />
</dynaGrid>
```

An application can change the size and contents of the initial dataSet object by defining a server-side callback method. You can specify the name of this callback method using the <dynaGrid> *OnCreateDataSet* attribute, as in the example above. The *OnCreateDataSet* value must be the name of a server-side callback method defined within the page class that contains the <dynaGrid> control. In the above example, this method name is **CreateDS**. The *OnCreateDataSet* callback method is called once, when the <dynaGrid> object is first created on the server and before the <dynaGrid> is first displayed.

The signature of the *OnCreateDataSet* callback method must look like this:

Class Member

```
Method CreateDS(
  pGrid As %ZEN.Component.dynaGrid,
  pDataSet As %ZEN.Auxiliary.dataSet) As %Status
{ }
```

Where:

- *pGrid* is the dynaGrid object that is invoking the callback.
- *pDataSet* is the dataSet object associated with the dynaGrid object.
- The method returns a %Status value indicating success or failure.

Typically a *OnCreateDataSet* callback method looks like this:

Class Member

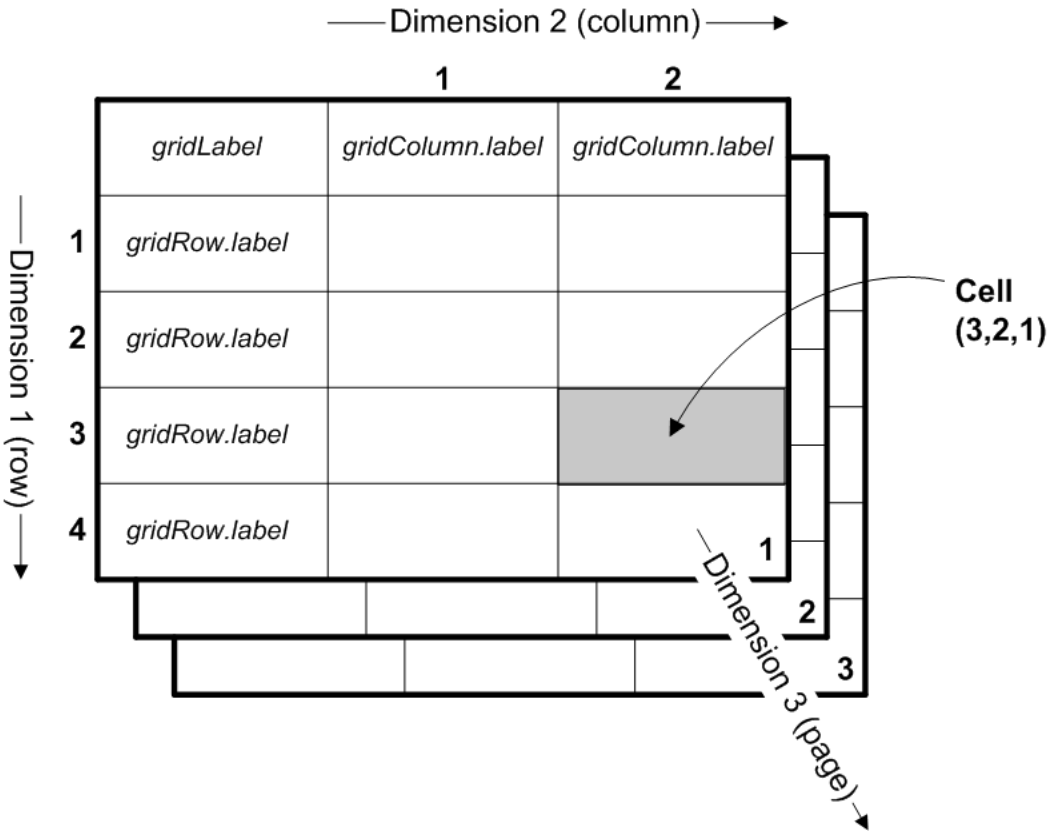
```
Method CreateDS(
  pGrid As %ZEN.Component.dynaGrid,
  pDataSet As %ZEN.Auxiliary.dataSet) As %Status
{
  // make sure dataSet is cleared out
  Do pDataSet.%Clear()

  // fill in contents of dataSet
  // This is a 2-D data structure

  // row labels (dimension 1)
  Do pDataSet.%SetLabel("Boston",1,1)
  Do pDataSet.%SetLabel("New York",2,1)
  Do pDataSet.%SetLabel("Chicago",3,1)
  Do pDataSet.%SetLabel("Miami",4,1)

  // column labels  (dimension 2)
  Do pDataSet.%SetLabel("Cars",1,2)
  Do pDataSet.%SetLabel("Trucks",2,2)
  Do pDataSet.%SetLabel("Trains",3,2)
  Do pDataSet.%SetLabel("Planes",4,2)

  // get size of dataSet
  Set rows = pDataSet.%GetDimSize(1)
  Set cols = pDataSet.%GetDimSize(2)
```

```
  // fill in initial data values
  For r=1:1:rows {
    For c=1:1:cols {
      Set value = 0
      Do pDataSet.%SetValue(value,r,c)
    }
  }

  Quit $$$OK
}
```

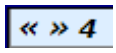
The above example defines a two–dimensional dataSet object with four rows and four columns. It supplies labels for the rows and columns and then loops over the cells to provide initial values for the cells.

If the *OnCreateDataSet* callback changes the dataSet object to contain three dimensions, this gives the <dynaGrid> the ability to move among "pages" of data. Each page is displayed as a two-dimensional grid that represents the currently visible page. The following figure illustrates this data model. For details regarding the label attributes shown in the figure, see the "<gridRow>," "<gridColumn>," and "<dynaGrid> Attributes" sections in this chapter.

Figure 5–1: Data Model for the Dynamic Grid Control



If there are pages in the <dynaGrid>, the *gridLabel* cell provides navigation information as follows. Click on the << symbol for the previous page or the >> symbol for the next page. The number indicates which page you are currently viewing:



5.9.1.2 <dynaGrid> Methods

Since the dataSet object is designed to work on either the server or the client, it provides local APIs for both environments. The following table lists the server-side (ObjectScript) methods. For further details, and to see the list of client-side (JavaScript) methods, see the class documentation for %ZEN.Auxiliary.dataSet.

Note: When you work with %ZEN.Component.dynaGrid programmatically, on the server side you can access the dataSet object via the dataSet property of the dynaGrid object. On the client, you cannot access the dataSet directly; you must the dynaGrid object's **getDataSet** method to get the dataSet object.

Server Method	Description
%Clear	Clear the contents of the dataSet (set every cell to "") without changing its size or number of dimensions.
%GetArray	Gets the contents of the dataSet as a multidimensional array, subscripted by the 1–based dimensional addresses of the cells (row, column, page). This array is passed to **%GetArray** by reference.
%GetDimensions	Returns the number of dimensions within the dataSet.
%GetDimSize	Receives a 1–based number identifying a dimension (row, column, page) and returns the number of cells in that dimension.
%GetLabel	Receives two numbers: • 1–based number identifying a cell position within a dimension • 1–based number identifying the dimension (row, column, page) **%GetLabel** returns the value of the label at the specified position.
%GetValue	Receives up to three 1–based numbers identifying a specific cell in the grid (row, column, page). Gets the value of the cell at that position.
%SetArray	Receives up to four arguments: • A multidimensional array, passed by reference, and subscripted by the 1–based dimensional addresses of the cells (row, column, page). The local array must have the same dimensionality as the dataSet and must have the correct number and type of subscripts. • Up to three 1–based numbers identifying the number of cells in each dimension of the grid (row, column, page). **%SetArray** copies the contents of the array into the dataSet.
%SetDimensions	Sets the number of dimensions within the dataSet to 1, 2, or 3.
%SetLabel	Receives three arguments: • String that specifies a label. • 1–based number identifying a cell position within a dimension • 1–based number identifying the dimension (row, column, page) **%SetLabel** copies the string to the label at the specified position.

Server Method	Description
%SetValue	Receives up to four arguments: • String that specifies a value. • Up to three 1–based numbers identifying a specific cell in the grid (row, column, page). **%SetValue** sets the value of the cell at the indicated position. **%SetValue** also updates the dimension size, if needed.

5.9.1.3 <gridRow>

A <dynaGrid> with dimensions greater than zero must contain one or more <gridRow> and <gridColumn> elements to define the initial dimensions of the grid. <gridRow> defines dimension 1 of a one-, two-, or three-dimensional <dynaGrid>.

The <gridRow> element is the XML projection of the %ZEN.Auxiliary.gridRow class. <gridRow> supports the attributes described in the following table.

Attribute	Description
height	HTML height value to apply to this row, such as 0.3in or 3%.
hidden	If true, this row is not displayed. The default is false. *hidden* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
label	Default text label for the row. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
readOnly	If true, cells in this row are read-only; the user cannot edit them. The default is false. *readOnly* has the underlying data type %ZEN.Datatype.boolean. See Zen Attribute Data Types.
rowName	Logical name of the row.
style	CSS style to apply to cells in this row, for example: `color: red;` If there is a column style and a row style active for a given cell, the row style is applied before the column style. This means the column style might override the row style.
title	Help text displayed when mouse hovers over this row. If there is a row title and a column title active for a given cell, the column title overrides the row title. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

When you work with %ZEN.Component.dynaGrid programmatically, you work with <gridRow> elements as members of the dynaGrid property called rows. This is a list collection of %ZEN.Auxiliary.gridRow objects. Each <gridRow> provided in the original <dynaGrid> definition in XData Contents becomes a member of the rows collection, associated with its ordinal position in the <dynaGrid>: 1, 2, 3, etc.

5.9.1.4 <gridColumn>

A <dynaGrid> with dimensions greater than zero must contain one or more <gridRow> and <gridColumn> elements to define the initial dimensions of the grid. <gridColumn> defines dimension 2 of a two- or three-dimensional <dynaGrid>.

The <gridColumn> element is the XML projection of the %ZEN.Auxiliary.gridColumn class. <gridColumn> supports the attributes described in the following table.

Attribute	Description
columnName	Logical name of the column.
hidden	If true, this column is not displayed. The default is false. *hidden* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
label	Default text label for the column. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
readOnly	If true, cells in this column are read-only; the user cannot edit them. The default is false. *readOnly* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
style	CSS style to apply to cells in this column, for example: `color: red;` If there is a row style and a column style active for a given cell, the row style is applied before the column style. This means the column style might override the row style.
title	Help text displayed when mouse hovers over this column. If there is a row title and a column title active for a given cell, the column title overrides the row title. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
width	HTML width value to apply to this column, such as 0.3in or 3%.

When you work with %ZEN.Component.dynaGrid programmatically, you work with <gridColumn> elements as members of the dynaGrid property called columns. This is a list collection of %ZEN.Auxiliary.gridColumn objects. Each <gridColumn> provided in the original <dynaGrid> definition in XData Contents becomes a member of the columns collection, associated with its ordinal position in the <dynaGrid>: 1, 2, 3, etc.

5.9.1.5 <dynaGrid> Attributes

When you place a <dynaGrid> element within an XData Contents block, you can assign it the following attributes. When working with the <dynaGrid> programmatically, these attributes are available as properties of the dynaGrid object.

Attribute	Description
Control component attributes	<dynaGrid> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes." Unlike most other controls, <dynaGrid> does not have a single *value*. Also, the <dynaGrid> width plays a special role in laying out the grid. For details, see the "<dynaGrid> Layout" section following this table.
columnWidth	Default width for columns that do not supply a width. You must specify *columnWidth* in pixels, for example 75px. The default is 100px. If no *columnWidth* is set and a <gridColumn> does not supply a *width*, the width of the column is calculated by dividing the remaining width of the grid amongst the columns with unspecified widths. For further details, see the "<dynaGrid> Layout" section following this table.
controllerId	If this <dynaGrid> is associated with a data controller, the *controllerId* attribute identifies the controller that provides the data for this <dynaGrid>. The *controllerId* value must match the *id* value provided for that <dataController>. See the chapter "Model View Controller."
currColumn	The 1–based column number of the currently selected cell in the grid. The default (until the user makes selections) is 1.
currPage	The 1–based page number of the data currently displayed in the grid. If the data set associated with this grid contains three–dimensional data, *currPage* indicates which page (along the third dimension) is currently displayed. The default is 1.
currRow	The 1–based row number of the currently selected cell in the grid. The default (until the user makes selections) is 1.
gridClass	CSS class to apply to the HTML table that displays the grid.
gridLabel	Caption to display in the upper-left corner cell. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
nowrap	If true, the contents of cells within the grid do not word wrap. If false, they do. The default is true. *nowrap* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
onchangecell	The *onchangecell* event handler for the <dynaGrid>. Zen invokes this handler whenever the user changes the contents of a cell. See "Zen Component Event Handlers." If you omit the *onchangecell* attribute, the default behavior is to assign the user-entered value to the cell.
onclickcolumn	Client-side JavaScript expression that is executed whenever the user clicks on a column header cell (at the top of a column).
onclickrow	Client-side JavaScript expression that is executed whenever the user clicks on a row header cell (at the left of a row).

Attribute	Description
OnCreateDataSet	The *OnCreateDataSet* value is the name of a server-side callback method that provides the data set associated with this <dynaGrid>. For details, see the section "<dynaGrid> Data Set."
ondblclick	Client-side JavaScript expression that is executed whenever the user double-clicks on the <dynaGrid>.
ondrawcell	Client-side JavaScript expression that is executed when the <dynaGrid> is about to draw a cell. If this event handler returns a value, this value is used as DHTML to render the cell contents. This convention provides a way to display custom cell formatting within a <dynaGrid>.
oneditcell	Client-side JavaScript expression that is executed when the <dynaGrid> is ready to accept user input in one of its cells. If the *oneditcell* event handler returns a value, this value is used as DHTML to render the editor used for the cell. *oneditcell* provides a way to display a custom cell editor within a <dynaGrid>. The default behavior is to place an HTML input control over that region of the grid to permit the user to enter data. There is no need to override this default, but you may.
onnotifyView	The *onnotifyView* event handler for the <dynaGrid>. See "Zen Component Event Handlers." This attribute applies if the <dynaGrid> is associated with a data controller. Zen invokes this handler each time the data controller connected to this <dynaGrid> raises an event. See the chapter "Model View Controller."
onselectcell	Client-side JavaScript expression that is executed whenever the user navigates to a new cell. The current cell row and column numbers are updated before the *onselectcell* call is made. The event handler is passed two variables, *row* and *col*, which contain the 1-based row and column number of the current cell.
rowLabelWidth	*rowLabelWidth* applies to the column of row labels. You must specify *rowLabelWidth* in pixels, for example 75px. The default is 100px. For further details, see the "<dynaGrid> Layout" section following this table.
scrollIntoView	If true, Zen uses the JavaScript scrollIntoView function to try and make visible the currently selected item within the grid. *scrollIntoView* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showColumnLabels	If true, column labels are displayed in a row along the top of the grid. The default is true. *showColumnLabels* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showRowLabels	If true, row labels are displayed in a column along the left side of the grid. The default is true. *showRowLabels* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

5.9.1.6 <dynaGrid> Layout

If the default layout of the <dynaGrid> does not suit your needs, you can adjust it. Laying out a <dynaGrid> involves many factors. These factors interact in ways that can counterbalance each other and produce surprising results. This section explains the underlying layout scheme for <dynaGrid> and the factors that you can control to produce the desired layout on the page.

The dimensions that you specify for the <dynaGrid> reserve space on the page in which to view the contents of the grid. The contents of the grid consist of the <dynaGrid> rows and columns. In practice, the contents may be larger, smaller, or the same size as the viewing space that you have allowed. If they are larger or smaller, the viewing space may resize itself to fit them. Alternatively, the viewing space may remain static and therefore "crop" or "mat" the grid contents. These options are under your control.

The primary interaction in <dynaGrid> layout is between the following two factors:

- The width specified for the <dynaGrid> reserves a viewing space on the page.
- The widths specified for columns within the <dynaGrid> lay out the grid contents relative to the viewing space.

Space Reserved to Display the <dynaGrid>

The dimensions that you specify using <dynaGrid> attributes specify the size of the enclosing HTML that Zen generates from your <dynaGrid> specification. The size of this is the size of the viewing space for the <dynaGrid>. You can specify this size by providing a *width* attribute for the <dynaGrid>, specifying a *width* attribute in the related CSS section, or providing a width value in an *enclosingStyle* attribute for the <dynaGrid>.

The resulting <dynaGrid> layout depends on whether the width is set to "auto" or to some fixed width. There is one behavior for "auto" and several behaviors for fixed widths depending on how the widths are set. If you do not set a width for the <dynaGrid>, the width defaults to "auto".

The layout variations for <dynaGrid> width settings are as follows:

- "auto" — The viewing space expands or shrinks to match the width of the grid contents.
- Fixed width — There are three cases:
  - If the <dynaGrid> contents turn out to be larger than the fixed viewing space, the viewing space size does not expand to fit the contents. Zen adds scroll bars to grant access to hidden parts of the grid.
  - If the <dynaGrid> contents turn out to be smaller than the fixed viewing space, the viewing space size does not shrink to fit the contents. Zen leaves any excess space blank.
  - If all <gridColumn> *width* values are specified as percentages, the columns expand or shrink proportionally, to exactly match the size of the viewing space.

Width of <dynaGrid> Data Columns

You can control the layout of <dynaGrid> contents by specifying column widths using the <dynaGrid> attributes *rowLabelWidth* and *columnWidth* and the <gridColumn> attribute *width*. The following sections explain how the values you choose for these attributes affect the layout results.

<dynaGrid> rowLabelWidth

rowLabelWidth applies to the column of row labels that appears as the leftmost column in the grid. The column of row labels acts as a vertical header for the grid, and so it is not treated like <gridColumn> elements, which contain grid data.

Unlike the <gridColumn> *width* attribute, the <dynaGrid> *rowLabelWidth* does not support percentage values. You must specify *rowLabelWidth* in pixels, for example 75px. The default is 100px.

<dynaGrid> columnWidth

columnWidth gives a default width for columns that do not supply a width. The <dynaGrid> *columnWidth* does not support percentage values. You must specify *columnWidth* in pixels, for example 75px. The default is 100px.

columnWidth is essential to the default calculation when a <dynaGrid> uses the default width ("auto") and specifies all of its <gridColumn> *width* values as percentages. In this case, Zen computes a display width for the data columns using *columnWidth* as described in the "<gridColumn> width" section.

The total width of a <dynaGrid> is the total width of all data columns plus whatever number of pixels was specified for *rowLabelWidth*, plus the space needed to render any specified margins, borders or cell padding. The <dynaGrid> viewing space sizes itself to display the entire grid.

<gridColumn> width

The <gridColumn> *width* attribute determines the width of individual data columns and may be specified in pixels (px) or percentages (%). Pixels and percentages may be freely mixed among the <gridColumn> elements in a given <dynaGrid> specification, with the following results:

- A column width in pixels is always respected.
- If some columns are specified in percentages and others in pixels, Zen uses the size of the pixel-based columns to calculate the width of the remaining columns.
- If all column widths are given as percentages and the <dynaGrid> has been given a fixed size, the grid performs a scale-to-fit operation such that the grid, any associate headers, borders, and column data exactly fill the defined viewing space.

If all the column widths are given as percentages and the <dynaGrid> has "auto" size or no size specified at all, the narrowest column is assigned the width of the <dynaGrid> *columnWidth*, and all other columns are sized with respect to the size of this column, as follows:

1. Find the <gridColumn> with the smallest percentage value. This is the narrowest column.
2. Consult the <dynaGrid> *columnWidth* value, assign this number of pixels as the width of the narrowest column, then use this result to calculate the number of pixels that corresponds to 1%.

   For example, if the smallest <gridColumn> *width* is 5% and the <dynaGrid> *columnWidth* is 75px, Zen calculates that there are 15 pixels in every 1%, for a total width of 150 pixels across all data columns in this grid.
3. Convert the percentage values for all other data columns into numbers of pixels.

5.9.2 <dataGrid>

<dataGrid> implements a component for viewing and editing tabular data. This is an HTML5 component. It works correctly only on HTML5 compliant browsers.

<dataGrid> has the following attributes:

Attribute	Description
Control component attributes	<dataGrid> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes."
canResizeColumns	If true, the user can use the cursor to resize columns. The default value is true. Has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
cellHoverColor	Specifies the background color of cells when the cursor hovers over them. The default value is #EEEEEE. See "Data Grid Layout."

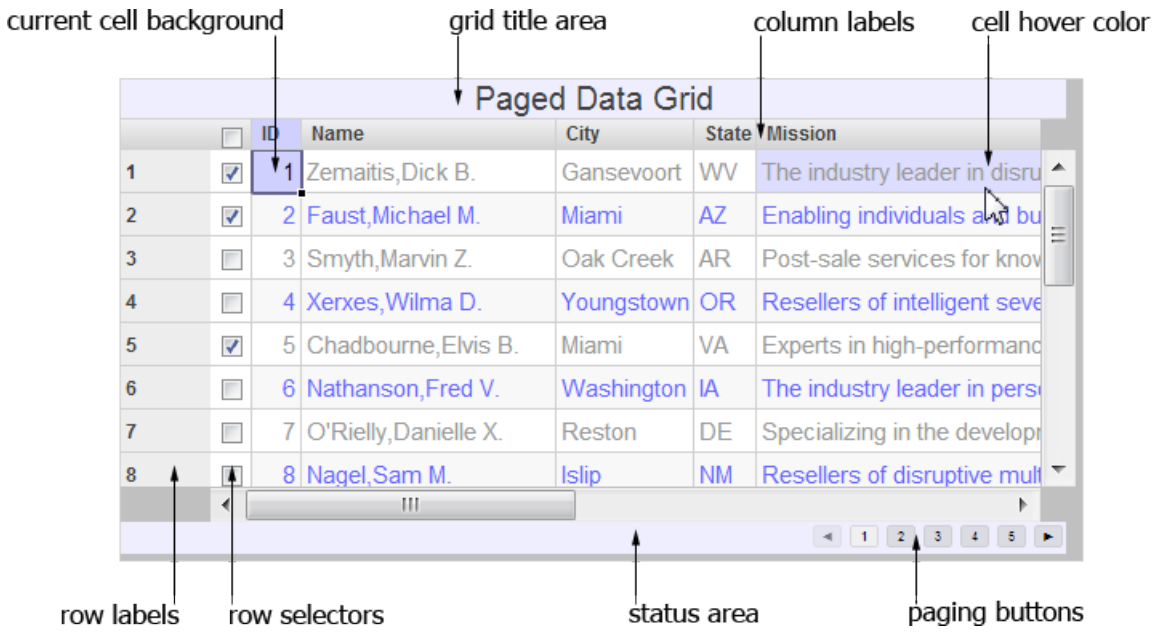
Attribute	Description
checkedRows	A string containing comma separated values. Each value in the string specifies the 1-based row number of a row in which the row-selector check box is selected. If all rows are checked, the value is set to "all". The property *showRowSelector* must be “true” for the user to be able to see and use the check boxes.
columnHeaderStyle	Additional CSS style to apply to column headers in this grid. The property *showColumnLabels* controls visibility of column headers. You can use *columnHeaderStyle* to set a *height* for column headers that is independent of the row height set using *rowHeight* or *rowHeaderStyle*.
columnLabelSpan	Specifies how parent column labels with multiple child labels are displayed. If true, then one parent label is displayed for each set of children. If false, then the parent row label is repeated for each child. The default value is false. Has the underlying data type %ZEN.Datatype.boolean. See “Zen Attribute Data Types.”
columnWidth	Default width, in pixels, for columns that do not supply a width. If no *columnWidth* is set and a column does not supply a *width*, the width of the column is calculated by dividing the remaining width of the grid amongst the columns with unspecified widths.
currCellBackground	The background color to apply to the currently selected cell. The default value is #D0D0FF. See “Data Grid Layout.”
currCellColor	The foreground color to apply to the currently selected cell. The default value is #000000. See “Data Grid Layout.”
currColumn	The 1–based column number of the currently selected cell in the grid. The default, until the user makes selections, is 1.
currPage	The 1–based page number of the data page currently displayed in the grid. If the grid displays data on multiple pages, *currPage* indicates which page is currently displayed. The default value is 1. Use the *getCurrPage* method to view the value of *currPage*. The attribute *pageSize* specifies the number of rows on each page. See the section on “Paging.”.
currRow	The 1–based row number of the currently selected cell in the grid. The default, until the user makes selections, is 1.
evenRowBackground	The background color to apply to even rows when the attribute *showZebra* is true. The default value is "#F8F8F8". See “Data Grid Layout.”
evenRowColor	The foreground color to apply to even rows when the attribute *showZebra* is true. The default value is #000000. See “Data Grid Layout.”
filterKey	If supplied, this is a key used to filter results that are already on the client. See “Filtering.”
format	The default format to apply to cells in this grid. This is a DeepSee format string: e.g., "###.##". Row and column-level formatting overrides this format.

Attribute	Description
gridStatusStyle	An optional CSS style to apply to the grid status area. See "Data Grid Layout."
gridTitle	An optional title to display along top of grid.
gridTitleStyle	An optional CSS style to apply to grid title. See "Data Grid Layout."
multiSelect	If true, users can select a range of cells in the grid. The default value is false. Has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
oddRowBackground	The background color to apply to odd rows when the attribute *showZebra* is true. The default value is #FFFFFF. See "Data Grid Layout."
oddRowColor	The foreground color to apply to odd rows when the attribute *showZebra* is true. The default value is #000000. See "Data Grid Layout."
pageSize	The number or rows displayed on each page when the grid displays data on multiple pages. Use the *getPageSize* method to view the value of *pageSize*. The default value is 0, which displays all rows on a single page. See the section on "Paging."
pagingMode	Specifies whether data paging should occur on the server or on the client. The default value is "client".
rowHeaderStyle	An additional CSS style to apply to row headers in this grid. The property *showRowLabels* controls visibility of row headers. If this property is used to set *height*, and *rowHeight* is also set, the grid uses the larger of the two values.
rowHeight	The default height, in pixels, used for rows that do not supply a height. If not defined, then the height is calculated. If *height* is also set with *rowHeaderStyle*, the grid uses the larger of the two values.
rowLabelSpan	Specifies how parent row labels with multiple child labels are displayed. If true, then one parent label is displayed for each set of children. If false, then the parent row label is repeated for each child. The default value is true. Has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
rowLabelWidth	Specifies the width of the column of row labels. The number you provide is interpreted as a number of pixels. If you do not provide a value, the width is calculated.
selectedRange	This specifies the current selected range of cells as a comma-separated list of integers. The list is of the form: *startRow*, *startCol*, *endRow*, *endCol*. All cells numbers are 1-based. If the range is equal to "", then no cells are selected. This property is used only if the attribute *multiSelect* is true.
selectMode	Specifies how selection works within the grid. If the value is "rows", then the user selects entire rows at a time. If the value is "cells", then the user can select and move between individual cells. The default value is "rows".

Attribute	Description
showColumnLabels	If true, column labels are displayed in a row along the top of the grid. The default is true. See "Data Grid Layout." The default column labels are the column names as provided by the data source. You can use the *caption* property of <columnDescriptor> to specify column labels. Has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showRowLabels	If true, row labels are displayed in a column along the left side of the grid. The default is true. See "Data Grid Layout." The default row labels are 1–based row numbers assigned to the rows. You can use the *caption* property of <rowDescriptor> to specify row labels. Has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showRowSelector	If true, display a check box in each row to allow selection of the row. If *showColumnLabels* is also true, a check box in the left-most column of the grid lets the user select or de-select all rows at once. The default value is false. See "Data Grid Layout." Has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showZebra	If true, display the grid with "zebra striping", alternating colors applied to rows. The default value is true. See "Data Grid Layout." The attributes *oddRowBackground* and *evenRowBackground* specify the background colors used, and *oddRowColor* and *evenRowColor* specify the foreground colors used. Has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
sortColumn	For sortable tables, this attribute specifies the 1-based column number of the column used for sorting results. The direction of the sort is specified by *sortOrder*. The default value is 0, which means no column is sorted. See "Sorting."
sortMode	Indicates where columns are sorted. Possible values are "client" or "server". The default value is "client". As of this release, server-side sorting is not implemented.
sortOrder	For sortable tables, this attribute specifies the sort order for values in the column specified by *sortColumn*. Possible values are: `""` (unsorted), `"asc"` (ascending), or `"desc"` (descending). The default value is unsorted. See "Sorting."
style	An additional style to apply to cells in this grid. This is applied before any row and column-level styles.
summaryRow	Optional descriptor for the summary row.
useEngine	Setting this property to 0 disables code execution when loading and rendering the grid, which is useful when you want to enter the equals sign (=) and not a formula in a particular column. This is a grid-wide setting.
valueColumn	The 1-based column number of the column that supplies the value for a row in the table. The default value is 1.

The following figure shows the major components of a <dataGrid>. In addition to the labeled items, the figure illustrates odd and even color and background color.

Figure 5–2: Data Grid Layout



The following attributes specify event handlers. See “Zen Component Event Handlers.”

Attribute	Description

Attribute	Description
onaction	A client-side JavaScript expression that is executed when the user executes an action within a cell, such as clicking on a column-defined checkbox, button, or link. Generally this expression invokes a client-side JavaScript method defined in the page class. This method becomes the "*onaction* event handler" for the grid. The current cell row and column number are updated before this call is made. The event handler is passed 3 variables, *row*, *name*, and *value*, which contain the row number of the current cell (1-based), the logical name of the column, and the current value of the action control, if applicable.
onchangecell	The *onchangecell* event handler for the grid. Zen invokes this handler whenever the user finishes editing a cell value. See "Zen Component Event Handlers." The current cell row and column number is updated before this call is made. The event handler is passed the new cell value. It should return the value to be placed into the cell or null to cancel the edit.
ondrawcell	The *ondrawcell* event handler for the grid. Zen invokes this handler whenever a cell is about to be drawn. See "Zen Component Event Handlers." The event handler is passed *value*, *row*, and *col* (1-based). If this event handler returns a value, then it is used to render the cell contents. The return value of this event handler is either null, in which case the default rendering is used for the cell, or an object with any of the following properties: • content - HTML to display within the cell. • align - horizontal alignment for the cell. • style - CSS style for the cell. • format - format string for the cell (ignored if content is supplied). • image - image for the cell (ignored if content is supplied). • type - type for the cell (ignored if content is supplied). • value - value for the cell (ignored if content is supplied).
onfiltercell	The *onfiltercell* event handler for the grid. Zen invokes this handler when the user enters a new *filterKey*. The event handler is passed an object, *info*, with the properties *row*, *col*, *value*, and *key*. The event handler should return true if the row containing the cell matches the filter, or false otherwise.
ongetlookupdata	The *ongetlookupdata* event handler for the grid. It returns a JavaScript array of data to display within the popup for a lookup column. The returned value can be any array of object or literal values.
ongetstatus	The *ongetstatus* event handler for the grid. If defined, this event handler returns the HTML that is displayed in the status area of this grid. The status area is located at the bottom of grid.
ongettitle	The *ongettitle* event handler for the grid. If defined, this event handler returns the HTML that is displayed in the title area of this grid. The title area is located at the top of grid. This supercedes the *gridTitle* property if defined.

Attribute	Description
onheaderclick	The *onheaderclick* event handler for the grid. If defined, this event is fired when the user clicks on a row or column header. The variable *which* indicates which header is clicked: "row" or "column". The variable *index* indicates the ordinal number of the header (1-based).
onselectcell	The client-side JavaScript expression that is executed whenever the user navigates to a new cell. The current cell row and column numbers are updated before the *onselectcell* call is made. The event handler is passed two variables, *row* and *col*, which contain the 1-based row and column number of the current cell. See "Selecting Rows and Cells."
onselectrow	The *onselectrow* event handler for the grid. If defined, this event is fired when the user toggles any of the visible row selector check boxes in the grid. A single variable, *range*, is passed to the event handler. The *range* is a string enumerating the currently checked rows. This variable has two special values: `""` indicating that no rows are currently checked, and the reserved value `"all"` indicating that all rows are checked. For values between these extremes, the range parameter is a CSV string listing the (1-based) indices of the currently checked rows. Because this event is linked to the toggling of the row selectors, it only fires if *showRowSelector* is true. See "Selecting Rows and Cells."

5.9.2.1 Creating a Data Grid

You can create an empty <dataGrid> control by simply adding the component to the Zen page:

```
<dataGrid id="grid"/>
```

You can control the size of the <dataGrid> by putting CSS styles in the XData Style section of the page. For example, if the grid *id* is 'grid':

```
#grid
{
  width: 600px;
  height: 600px;
}
```

You can also control the size of the grid using JavaScript by setting the width and height of the grid's enclosing div and then invoking the grid's **adjustSizes** method to lay out the internal parts of the grid:

```
zen('grid').adjustSizes();
```

5.9.2.2 Connecting the Grid to Data

The <dataGrid> is a data view component and can be connected to an <altJSONProvider>. See the section "Zen JSON Components" in the book *Developing Zen Applications*. For example, if you have an <altJSONProvider> that supplies an array of objects:

XML

```
<altJSONProvider id="json" OnGetArray="GetData" />
```

You can connect the <dataGrid> to this <altJSONProvider> by setting the *controllerId*:

```
<dataGrid id="grid" controllerId="json" />
```

The grid now displays the data served by the <altJSONProvider>. In this case, all the properties provided by the <altJSON-Provider> are displayed as columns and there is one row for every object (data series) supplied by the provider.

5.9.2.3 Filtering

<dataGrid> supports client-side filtering of the grid contents. Filtering limits the data actually displayed by applying a filtering function to the client-side data before displaying it in the grid. You can override the default filtering behavior by overriding the *onfiltercell* event handler.

5.9.2.4 Searching

<dataGrid> supports server-side searching. The search is performed by changing the parameters used to fetch the data displayed in the grid. Typically this is done by reloading the contents of an <altJSONProvider>.

5.9.2.5 Sorting

A <dataGrid> supports column-wise sorting on the client. The attribute *sortColumn* specifies the column used for sorting. The attribute *sortOrder* determines the order in which the column should be sorted. If *showColumnLabels* is true, the column label for the sorted column displays an arrow that indicates the direction of the sort, as illustrated in the following figures.

Figure 5–3: Ascending and Descending Sort Order

The attribute *sortMode* can have a value of either "client" or "server," indicating client-side or server-side sorting. The default value is "client." As of this release, server-side sorting is not implemented.

5.9.2.6 Paging

The <dataGrid> component supports paging. Each page of the grid displays a fixed set of rows. Paging can be done on either the client or the server. If paging is done on the client, the grid loads all the rows and the user pages through them on the client. If paging is done on the server, the grid loads one page at a time from the server.

To use client-side paging:

- Set the <dataGrid> *pagingMode* attribute to "client":

  ```
  <dataGrid pagingMode="client" />
  ```

- Set the <dataGrid> *pageSize* attribute to the number of rows to show per page:

  ```
  <dataGrid="10"/>
  ```

If the number of rows displayed by the grid is greater than the number specified by *pageSize*, a set of paging buttons is displayed on the bottom of the grid. See "Data Grid Layout."

To use server-side paging:

- Set the <dataGrid> pagingMode attribute to "server":

  ```
  <dataGrid pagingMode="server" />
  ```

- The <dataGrid> *pageSize* attribute is ignored. Instead, set the *pageSize* attribute of the <altJSONSQLProvider>.

5.9.2.7 Selecting Rows and Cells

The properties *onselectcell* and *onselectrow* let you define an event handlers that run when the end user selects a row or cell in the <dataGrid>. For example, a Zen page containing an <altJSONProvider> and <dataGrid> as follows:

XML

```
<page xmlns="http://www.intersystems.com/zen">
  <altJSONProvider id="json" OnGetArray="GetData" />
  <dataGrid id="grid"
    controllerId="json"
    showRowSelector="true"
    onselectrow="zenPage.rowSelected(range);" />
</page>
```

And a `rowSelected` callback defined as follows:

Class Member

```
ClientMethod rowSelected(range) [ Language = javascript ]
{
  alert("Selected rows are: "+range);
}
```

Pops up an alert message showing the currently selected rows. Similarly, the following page:

XML

```
<page xmlns="http://www.intersystems.com/zen">
  <altJSONProvider id="json" OnGetArray="GetData" />
  <dataGrid id="grid"
    controllerId="json"
    multiSelect="true"
    selectMode="cells"
    onselectcell="zenPage.cellSelected(row, col);" />
</page>
```

And a `cellSelected` callback defined as follows:

Class Member

```
ClientMethod cellSelected(row, col) [ Language = javascript ]
{
  alert("Selected cell is: "+ row + ", " + col);
  var rng = zen('grid').selectedRange
  alert("Selected range is: " + rng);
}
```

5.9.2.8 Column Descriptors

You can exercise more direct control over the contents of the grid by defining one or more <columnDescriptor> objects. If the grid contains <columnDescriptor> objects, it no longer automatically displays all of the data supplied by its data controller. You must provide a <columnDescriptor> for each column you want in the grid. The *value* property specifies the data shown in each column.

XML

```
<dataGrid id="grid" controllerId="json">
 <columnDescriptor value="100"/>
 <columnDescriptor value="=[@Name]"/>
 <columnDescriptor value="=[@City]"/>
</dataGrid>
```

The grid in this example shows 3 columns. The first column displays the value "100" in each cell. The second column displays the value of the *Name* property supplied by the altJSONProvider. The "=" character at the start indicates that this is an expression (or formula). The [ ] brackets enclose an identifier. The identifier "@Name" specifies the *Name* property supplied by the data controller.

<columnDescriptor> has the following attributes:

Attribute	Description
align	Optional horizontal alignment to apply to this column. Use this to control alignment rather than the *style* property.
caption	Optional caption to apply to this element. It appears in the grid as the column label text.
columns	Optional child descriptors for this column.
format	Optional format to apply to this element.
headerAlign	Optional horizontal alignment to apply to the header for this column. If not defined, then the value specified in *align* is used. Use this to control alignment rather than the *style* property.
headerStyle	Style string to apply to the header for this column.
hidden	Do not show this column.
image	For image columns, this is the name of the image to display.
ongetlookupspec	This event is used to compute the lookup (popup) information for this column.
priority	Optional priority to apply to this element.
readOnly	Optional readOnly attribute to apply to this element. If true, the contents of this column cannot be edited. The default value is false.
style	Style string for this column.

Attribute	Description
type	If the <dataGrid> *selectMode* property is set to "cells", then you can use the *type* property of the column descriptor to specify the type of edit control that is displayed in the cell when it is selected. Possible values for the *type* property are: • "string" — display a string value and edit as a string. • "image" — display an image (using the url in the image property). • "button" — display a button. You must implement an *onaction* handler to supply behavior for the button. • "checkbox" — display a checkbox. • "link" — display an HTML link. You must implement an *onaction* handler to supply behavior for the link. • "lookup" — display a lookup control. You must implement an *ongetlookupdata* handler to supply behavior for the lookup. • "user" — display arbitrary HTML as defined by the *ondrawcell* callback
value	Optional default value for this column. This can be a literal value or an expression such as this one: "=[@Name]" The "=" character at the start indicates that this is an expression (or formula). The [ ] brackets enclose an identifier. The identifier "@Name" specifies the *Name* property supplied by the data controller. If you do not specify the *value* attribute, the grid displays values from the data controller in the order in which they are supplied.
width	Optional default minimum width, in pixels, used for this column on initial rendering. The actual width may be adjusted higher as needed depending on the actual widths of data in the cells themselves. This setting does not preclude the end user from manually resizing the column width to a smaller value.

You can use CSS to modify non-essential display properties of editable cells in a <dataGrid>. The CSS class for editable cells is `dgCellEditor`. For example, the following CSS sets the cell background to yellow during editing:

```
.dgCellEditor {
  background-color:yellow;
}
```

You can also modify display properties of lookup cells, specifically the image on the button the user clicks to open the lookup list. The CSS class is `dgAction`. The URL specified for an image file is resolved with respect to the csp/broker directory under the Caché install directory. For example, the following CSS sets the lookup button image:

```
.dgAction {
  background-image:url("lookupicon.png");
}
```

5.9.2.9 Row Descriptors

You can exercise more direct control over the contents of the grid by defining one or more <rowDescriptor> objects. If the grid contains <rowDescriptor> objects, it no longer automatically displays what is served by its data controller. You must provide a <rowDescriptor> for each row you want in the grid.

XML

```
<dataGrid id="grid" controllerId="json">
  <rowDescriptor caption="ROW 1"/>
  <rowDescriptor caption="ROW 2"/>
  <rowDescriptor caption="ROW 3"/>
</dataGrid>
```

<rowDescriptor> has the following attributes:

Attribute	Description
caption	Optional caption to apply to this element. It appears in the grid as the row label text.
format	Optional format to apply to this element.
priority	Optional priority to apply to this element.
readOnly	Optional readOnly attribute to apply to this element. If true, the contents of this row cannot be edited. The default value is false.
rows	Optional child descriptors for this row.
style	Style string for this column.

5.10 Hidden

The Zen <hidden> control is a wrapper around the HTML `<input type="hidden">` element. The <hidden> control is present in the form, and has a *value* associated with it, but is never visible to the user. The *value* can be changed programmatically on the client or server side. When the form is submitted, the values of any <hidden> controls are submitted along with all the others.

<hidden> has the same general-purpose attributes as any Zen control. For descriptions, see the section "Control Attributes."

6

Model View Controller

Model View Controller, or *MVC*, is a well known architecture for user interface design. This chapter describes how Zen implements MVC, and how to add MVC features to a Zen page.

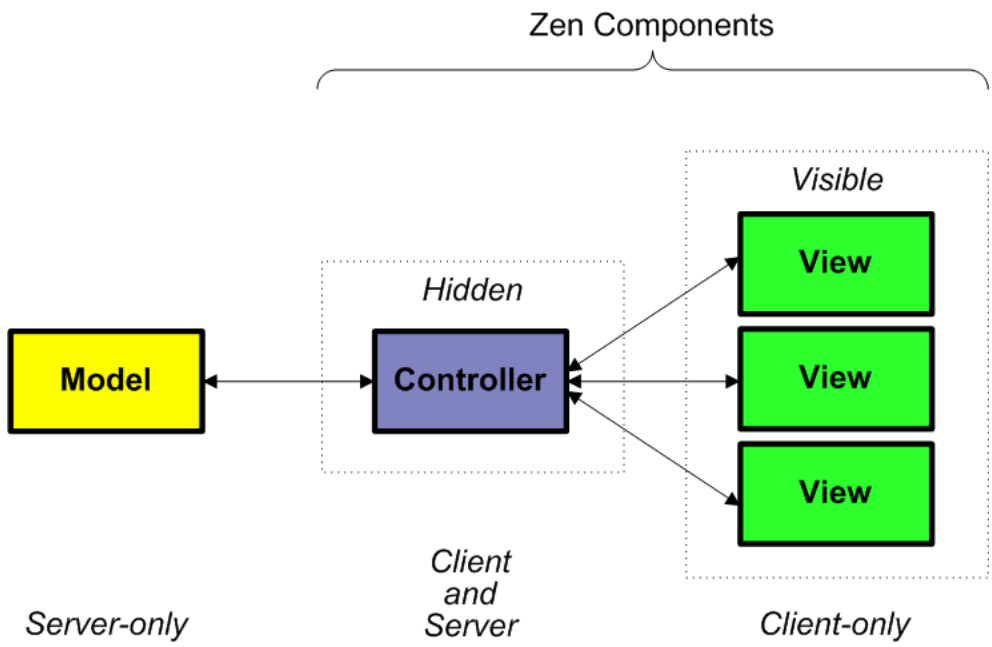
To simplify the flow of data from a data source to a Zen page, Zen provides a set of classes that let you define a data model (the *model*) and connect it to a set of Zen components (the *view*) via an intermediate object (the *controller*). When the model associated with a controller changes, these changes are automatically broadcast to all views connected to the controller.

The following are some typical uses of MVC:

- Create a form that displays a set of properties, loaded from a persistent object within the database. The form automatically displays controls appropriate to the data type of each property.
- Display a chart based on values within a form. The chart automatically updates its display whenever the user submits any changes to the form.
- Display meters representing values calculated on the server. When the page is refreshed, these meters automatically update themselves with current values from the server.

The following figure shows the three parts of the Zen MVC architecture — model, view, and controller — and indicates where these objects execute their code. The controller and its associated views are Zen components, placed on the Zen page. The controller component is hidden from view, but view components are user-visible. The view components display data values, which they obtain by requesting them from the controller. The controller resides on the client, but has the ability to execute code on the server. The model resides entirely on the server. It draws its data values from a source on the server and can respond to requests for data from the controller.

Figure 6–1: Model View Controller Architecture



The next three sections in this chapter expand the discussion of each part of the previous figure:

- "Model"
- "Controller"
- "View"

Remaining sections in this chapter provide a series of exercises that show how to use MVC features to create a form:

- "Constructing a Model"
- "Binding a <form> to an Object Data Model"
- "Adding Behavior to the <form>"
- "<dynaForm> with an Object Data Model"
- "<dynaForm> with an Adaptor Data Model"

6.1 Model

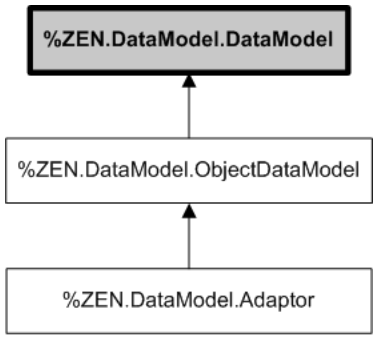
A data model is any subclass of %ZEN.DataModel.DataModel. A data model can:

1. Retrieve data values from one or more sources, such as:
   - A persistent Caché object
   - An external database (ODBC or JDBC) via the Caché SQL Gateway
   - A Caché global
   - An Ensemble business service
2. Place these values into its own properties.

3. Make these properties available to be consumed by a data controller.

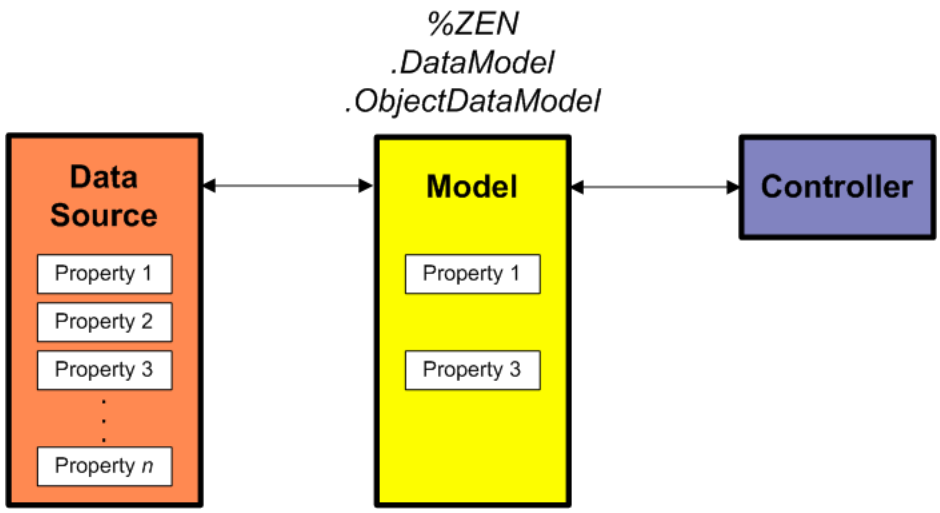
There are two variations on a data model class. Typically you choose one of these as your parent class when you create a new data model. The variations are closely related, but serve different purposes. The available data model subclasses are %ZEN.DataModel.ObjectDataModel and %ZEN.DataModel.Adaptor, as shown in the following figure.

Figure 6–2: Data Model Classes



6.1.1 %ZEN.DataModel.ObjectDataModel

A subclass of %ZEN.DataModel.ObjectDataModel is called an *object data model*. It defines one or more properties that a data controller component can consume. Each of these properties is correlated with a value from the data source class. Not every value in the data source needs to be exposed in the model. This convention allows you to expose in the data model only those values that you wish to.



An example of this might be a patient record, which contains confidential information that not every application should expose. Keep in mind that when you use this option, the developer of the object data model class is responsible for implementing methods to load values from a source into data model properties, store values back to a source, and validate values. This is in contrast to the adaptor data model, where Zen takes care of these details. However, this is not such a difficult procedure.

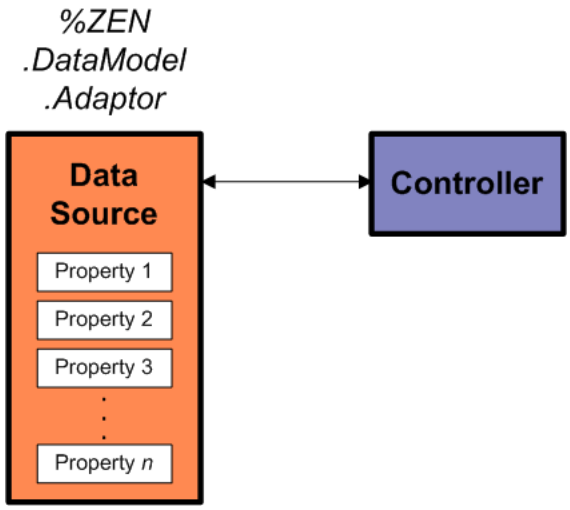
For more information about using %ZEN.DataModel.ObjectDataModel, consult the following sources:

- For step by step instructions, see the section "Binding a <form> to an Object Data Model."
- For code examples, use Studio to view the classes ZENMVC.FormDataModel and ZENMVC.FormDataModel2 in the "SAMPLES" namespace.

- For reference information, see the section "Data Model Classes."

6.1.2 %ZEN.DataModel.Adaptor

There are many times when it is convenient to use a persistent object as a data model. To make this easy to accomplish, Zen provides the %ZEN.DataModel.Adaptor interface. Adding this class as an additional superclass to a persistent class makes it possible to use the persistent class as a data model. This data model makes available to a data controller any and all properties that it contains. This option is useful when you want to expose every property in an existing class.



An example of this might be a class that you are using in an inventory or parts control application, wherein each product might be described by a class with a large number of properties. If you want to place all of these properties onto a form automatically without writing another class, you can simply cause the product class to extend %ZEN.DataModel.Adaptor. Following that, Zen simply generates the form for you, as later topics explain. The drawback of this choice is that your data and form are very closely linked. If you want some flexibility, you should subclass the object data model class and implement the internal interface. This is the classic trade-off of convenience versus flexibility.

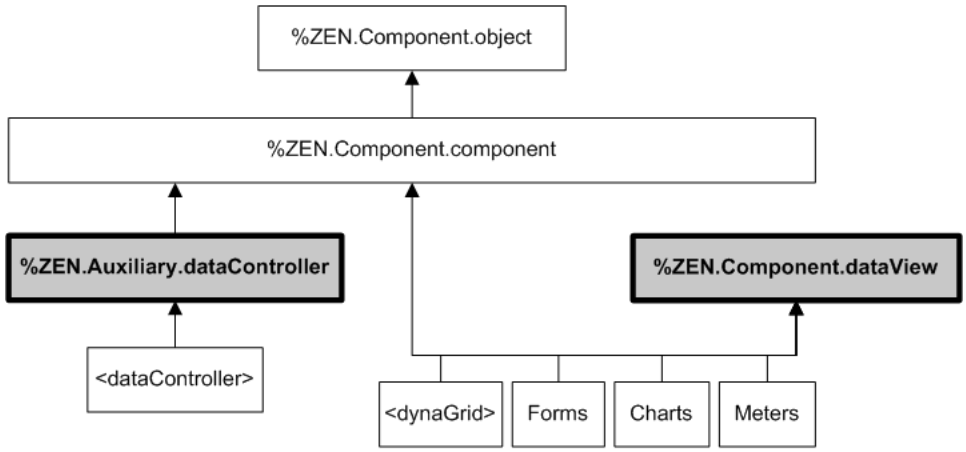
For more information about using %ZEN.DataModel.Adaptor, consult the following sources:

- For step by step instructions, see the section "<dynaForm> with an Adaptor Data Model."
- For code examples, use Studio to view the classes ZENMVC.Address and ZENMVC.Person in the "SAMPLES" namespace.
- For reference information, see the section "Data Model Classes."

6.2 Controller

A data controller manages the communication between a data model and a data view. A data controller is any subclass of %ZEN.Auxiliary.dataController. Through class inheritance, every data controller is also a Zen component, as the following figure shows. This convention permits you to place a data controller on a Zen page.

Figure 6–3: Data Controller and Data View Classes



6.2.1 <dataController>

To provide a data controller for a Zen page, simply add a <dataController> or a subclass of %ZEN.Auxiliary.dataController inside the <page>. The <dataController> component appears in XData Contents along with other components, but it is not visible onscreen. It acts as an intermediary between a data model and one or more data views.

The following example defines a <dataController> that opens an instance of the class MyApp.MyModel using an *id* value of 1. A <dynaForm> is bound to the <dataController> by setting the <dynaForm> *controllerId* property to the *id* of the <dataController>. This causes the <dynaForm> to display a form that provides a Zen control for every property within the *modelClass*.

```
<dataController id="data" modelClass="MyApp.MyModel" modelId="1"/>
<dynaForm id="myForm" controllerId="data"/>
```

6.2.2 <dataController> Attributes

When you place a <dataController> within a Zen <page>, you can assign it the following attributes:

Attribute	Description
Zen component attributes	A <dataController> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen* The *id* attribute is required for <dataController>. *name* and *condition* may also apply. A <dataController> is not visible, so visual style attributes do not apply.
alertOnError	If true, the <dataController> displays an alert box when it encounters errors while invoking server-side functions, such as when saving or deleting. The default is true. *alertOnError* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

Attribute	Description
autoRefresh	Setting *autoRefresh* to a non-zero value turns on automatic refresh mode for this data controller. In this mode, the data controller reloads its data from the server at the periodic interval specified by *autoRefresh* (in milliseconds). *autoRefresh* is provided as a convenience for data controller used to drive meters or charts; it is of limited use for forms. Setting *autoRefresh* to 0 disables automatic refresh mode.
defaultSeries	Optional. If a data model has multiple data series, *defaultSeries* is a 1-based number that specifies which series should be used to provide values to data views that can only display values from one data series (such as a form). The default is 1.
modelClass	Package and class name of the data model class that provides data for this <dataController>. The *modelClass* value can be a literal string, or it can contain a Zen #()# runtime expression.
modelId	String that identifies a specific instance of a data model object. The form and possible values of the *modelId* string are determined by the developer of the data model class. The *modelId* value can be a literal string, or it can contain a Zen #()# runtime expression.
oncreate	The *oncreate* event handler for the <dataController>. Zen invokes this handler each time the **createNewObject** method is called. See "Zen Component Event Handlers."
ondelete	Client-side JavaScript expression that runs each time the **deleteId** method is called. The *ondelete* callback is invoked whether or not the delete succeeds. The *ondelete* callback can make use of two special variables: `id` contains the *modelId* of the deleted object, and `deleted` indicates whether or not the delete was successful.
onerror	Client-side JavaScript expression that runs each time the data controller attempts to open an instance of a data model object and encounters an error. When you work with a %ZEN.Auxiliary.dataController programmatically, you can obtain the most recent error message reported by the data model object associated with this data controller, by examining the modelError property of the dataController object. This property is not available as an XML attribute when adding the <dataController> to the Zen page. To access this property from the client side, use the data controller's client-side JavaScript method **getError**.
onnotifyController	Client-side JavaScript expression that runs each time a data view connected to this data controller raises an event.
onsave	Client-side JavaScript expression that runs each time the **save** method is called. The parameter `id` is passed to the event handler and contains the current *modelId*.
readOnly	If true, this data controller is read-only, regardless of whether or not its corresponding data model is read-only. The default is false. *readOnly* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

6.2.3 <dataController> Methods

A <dataController> or a subclass of %ZEN.Auxiliary.dataController provides the following client side JavaScript methods for working with the data controller and the data model that it represents. There are more methods described in the online Class Reference documentation for the %ZEN.Auxiliary.dataController class.

Client Side Method	Purpose
getDataAsArrays()	Returns the data in this controller as an array of arrays. This is useful when working with charts.
getDataAsObject(*series*)	Returns the data in this controller as an instance of a zenProxy object with properties whose names and values correspond to the properties of the current Data Model object. If the data model supports more than one data series, then *series* (0-based) specifies which series to use (the default is 0). For information about zenProxy, see Zen Proxy Objects in the "Zen Pages" chapter of *Developing Zen Applications*.
getDataByName(*prop*)	Returns the data in the specified *prop* (property) of this data controller object. This data is equivalent to the *value* of the corresponding control on the generated form.
getDimensions()	Return number of dimensions within the dataModel. There are 2 dimensions: The first is the set of properties, the second has a typical size of 1. The second dimension may be larger than 1 in cases where the model serves multiple series for a given model instance. (Such as when providing multiple data series for charts).
getDimSize(*dim*)	Return the number of items in the specified dimension *dim*. *dim* is 1,2, or 3.
getLabel(*n*,*dim*)	Get the label at position *n* in the given dimension *dim*. *n* is a 0–based number. *dim* is 1,2, or 3.
getModelClass()	Return the current *modelClass* value.
getModelId()	Return the current *modelId* value.
raiseDataChange()	Notify listeners that the data associated with this data controller has changed.
setDataByName(*p*,*v*,*s*)	Change the specified property *p* of this data controller object to the value *v*. This also changes the *value* of the corresponding control on the generated form. If *p* is "%id" change the *id* of this controller. If *p* is "%series" change the *defaultSeries* of this controller. If the data model supports more than one data series, then *s* (0-based) specifies which series to use (the default is 0). Following a call (or multiple calls) to **setDataByName**, you must subsequently call **raiseDataChange** to notify listeners that the data associated with this data controller has changed.
setModelId(*id*)	Change the *modelId* value at runtime. Changing the *modelId* value causes the controller to load a new record, and to update its associated views.

Client Side Method	Purpose
setModelClass(*name*)	Change the *modelClass* value at runtime. Optionally, you can call **setModelClass**(*name*,*id*) to change both the *modelClass* and the *modelId*. Changing the *modelClass* value causes the controller to abandon the previous model and load data from the new model into the controller.

6.3 View

A data view is any Zen component that implements the %ZEN.Component.dataView interface. The list of data view components includes:

- "<dynaGrid>"
- "All forms"
- "All charts"
- "All meters"

For an illustration, see the figure "Data Controller and Data View Classes" in the "Data Controller" section.

A data view component connects to its associated data controller at runtime, and uses it to get and set values from the associated data model. A data view points to its data controller; more than one data view can point to the same data controller.

6.3.1 Data View Attributes

Data view components support the usual Zen component attributes, plus any specialized attributes that are typical of the specific type of component. Additionally, all data view components support the following attributes, which relate specifically to the component's role as a data view.

Important: If a user sets a component's value by interacting with the Zen page, the controller and thus the model are notified. If program code sets the value, the controller is not notified, and the value is lost on submit. Program code should write directly to the controller, which then updates the control.

Attribute	Description
controllerId	Identifies the data controller for this data view (). The *controllerId* value must match the *id* value provided for that <dataController> component.
onnotifyView	The *onnotifyView* event handler for the data view component. Zen invokes this handler each time the data controller associated with this data view raises an event. See "Zen Component Event Handlers."

Meters and controls support the following property:

Attribute	Description
dataBinding	Identifies the data model property that is bound to this component. This property provides the value that the component displays: • If the *dataBinding* value is a simple property name, this is assumed to be a property within the data model class identified by the <dataController> *modelClass* attribute. • Alternatively, *dataBinding* can provide a full package, class, and property name. *dataBinding* is generally suitable for components that display a single value (meters or controls). Each meter on a Zen page must supply a *controllerId* and a *dataBinding*. Controls do not support the data view interface, so cannot supply a *controllerId*, but if the form that contains the controls has an associated data controller, each control within the form can supply a *dataBinding* attribute that identifies which property it displays.

6.3.2 The Controller Object

When you work with a data view component programmatically, you can obtain a reference to the associated %ZEN.Auxiliary.dataController object via the following properties of the %ZEN.Component.dataView interface:

- `controller` — Use in JavaScript code that runs on the client side
- `%controller` — Use in ObjectScript, Caché Basic, or Caché MVBasic code that runs on the server side

6.3.3 Multiple Data Views

Whenever a user modifies a value within one of the controls that is bound to a data controller, the data controller is notified. It is common for data views to share a controller; for example, different types of chart on the same page could share the same data controller to display different visualizations of the same data, as in the following figure. If there are multiple data view components connected to the same data controller, they are all notified of any change to a bound control.

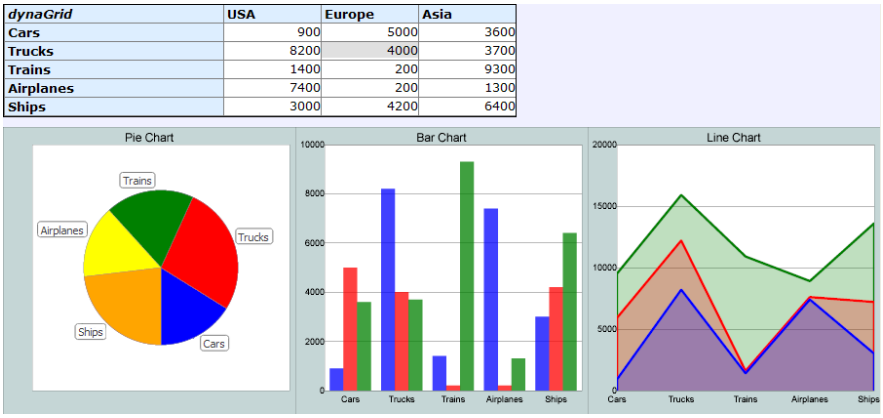
For examples of shared data controllers, use Studio to view the classes ZENMVC.MVCChart and ZENMVC.MVCMeters. The class ZENMVC.MVCMeters uses the same data controller to provide values for a <dynaGrid> and several meters. The class ZENMVC.MVCChart provides a <dynaGrid> and three charts that all use the same data controller. Try entering the following URIs in the browser:

`http://localhost:57772/csp/samples/ZENMVC.MVCChart.cls`

`http://localhost:57772/csp/samples/ZENMVC.MVCMeters.cls`

Where 57772 is the web server port number that you have assigned to Caché.

When you change a value in one of these pages by editing it in the <dynaGrid> and pressing **Enter**, this change affects the corresponding value in all the charts (or meters) on the same page, because all of them share the same data controller. In the following figure, the user has just modified the Trucks field in the <dynaGrid> for ZENMVC.MVCChart.



6.4 Constructing a Model

This topic provides the first in a series of exercises that show how to use the Model View Controller to create a form. If you have a new Caché installation, before you begin these exercises you must first run the ZENDemo home page. Loading this page silently generates data records for the SAMPLES namespace. You only need to do this once per Caché installation.

Enter the following URI in the browser:

```
http://localhost:57772/csp/samples/ZENDemo.Home.cls
```

Where 57772 is the web server port number that you have assigned to Caché.

6.4.1 Step 1: Type of Model

There are two basic choices when generating a form using the Model View Controller:

- %ZEN.DataModel.ObjectDataModel or %ZEN.DataModel.Adaptor

  An object data model takes more work to code. An object data model gives you explicit control over which properties end up in the model, so it makes more sense for cases when you want to shield certain properties from display or when you want to display charts, which require very fine data control. The choice of adaptor data model is simple and convenient, but it imposes a burden on the original persistent object, in that you must change its class code to allow it to become an adaptor data model.

- <form> or <dynaForm>

  A <form> is a good choice for the key forms that are critical to the success of your application. A <form> requires more work to encode, but provides as fine a level of control as you need to perfect the results. A <dynaForm> provides automatic results and automatically updates its layout if you change the underlying data model. This is useful to generate forms for large volumes of detailed information such as system administration, inventory, or maintenance requests.

For this exercise we choose an object data model and a <form>.

6.4.2 Step 2: Object Data Model

Suppose a persistent object called Patient contains a patient record. This object may have hundreds of properties. Suppose you want to create a simple page that only displays demographic information, in this case the patient's name and city of residence. This is a clear case for using %ZEN.DataModel.ObjectDataModel.

First, define an object data model class called PatientModel that knows how to load and store properties from the Patient object:

1. Start Studio.
2. Choose **File > Change Namespace** or **F4**.
3. Choose the SAMPLES namespace.
4. Choose **File > New** or **Ctrl-N** or the  icon.
5. Click the **General** tab.
6. Click the **Caché Class Definition** icon.
7. Click **OK**.
8. For **Package Name** enter:

   ```
   MyApp
   ```

9. In the **Class Name** field, type:

   ```
   PatientModel
   ```

10. Click **Next**.
11. Choose **Extends**
12. Click **Next** and enter (or browse to) this class name:

    ```
    %ZEN.DataModel.ObjectDataModel
    ```

13. Click **Finish**.

    Studio creates and displays a skeletal object data model class:

    ```
    Class MyApp.PatientModel Extends %ZEN.DataModel.ObjectDataModel
    {

    }
    ```

14. Add properties and methods to complete the class as shown in the following code example. This example defines two properties for the data model (Name and City). It also overrides several of the server-side methods in the %ZEN.DataModel.ObjectDataModel interface. For documentation of these methods, see the section "Object Data Model Callback Methods."

Class Definition

```
Class MyApp.PatientModel Extends %ZEN.DataModel.ObjectDataModel
{
  Property Name As %String;
  Property City As %String;

  /// Load an instance of a new (unsaved) source object for this DataModel.
  Method %OnNewSource(Output pSC As %Status = {$$$OK}) As %RegisteredObject
  {
    Quit ##class(ZENDemo.Data.Patient).%New()
  }

  /// Save instance of associated source object.
  Method %OnSaveSource(pSource As ZENDemo.Data.Patient) As %Status
  {
    Set tSC=pSource.%Save()
    If $$$ISOK(tSC) Set ..%id=pSource.%Id()
    Quit tSC
  }

  /// Load an instance of the source object for this DataModel.
  Method %OnOpenSource(pID As %String, pConcurrency As %Integer = -1,
                       Output pSC As %Status = {$$$OK}) As %RegisteredObject
  {
```

```
    Quit ##class(ZENDemo.Data.Patient).%OpenId(pID,pConcurrency,.pSC)
  }

  /// Delete instance of associated source object.
  ClassMethod %OnDeleteSource(pID As %String) As %Status
  {
   Quit ##class(ZENDemo.Data.Patient).%DeleteId(pID)
  }


  /// Do the actual work of loading values from the source object.
  Method %OnLoadModel(pSource As ZENDemo.Data.Patient) As %Status
  {
    Set ..Name = pSource.Name
    Set ..City = pSource.Home.City
    Quit $$$OK
  }

  /// Do the actual work of storing values into the source object.
  Method %OnStoreModel(pSource As ZENDemo.Data.Patient) As %Status
  {
    Set pSource.Name = ..Name
    Set pSource.Home.City = ..City
    Quit $$$OK
  }
}
```

15. Choose **Build > Compile** or **Ctrl-F7** or the  icon.

6.5 Binding a <form> to an Object Data Model

This exercise creates a data controller based on the object data model from the previous exercise, "Constructing a Model." It then binds a form to this data controller.

6.5.1 Step 1: Data Controller

If you do not already have a simple Zen application and page class available from previous exercises, create them now using instructions from the "Zen Tutorial" chapter in *Using Zen*:

- "Creating a Zen Application" — Create the class MyApp.MyNewApp.
- "Creating a Zen Page" — Create the class MyApp.MyNewPage. You only need to complete the steps in the first section, "Step 1: New Page Wizard."

Now place a data controller component on the Zen page by adding a <dataController> element in the MyApp.MyNewPage class XData Contents block, inside the <page> container, as follows:

XML

```
<dataController id="patientData"
                modelClass="MyApp.PatientModel"
                modelId="1" />
```

Where:

- *id* is the unique identifier of the data controller on the Zen page. A data view (such as a form) specifies its data controller using this *id*.
- *modelClass* is package and class name of the data model class. The previous exercise, "Constructing a Model," created the class MyApp.PatientModel which is an object data model that represents objects of the ZENDemo.Data.Patient class.
- *modelId* is a number that identifies the record to initially load from the data source. In this case, it is the identifier of a specific ZENDemo.Data.Patient object.

Note: The <dataController> component is not visible on the page.

6.5.2 Step 2: Data View

Now create a form and connect it to the data controller, as follows:

1. Place a <form> component inside the <page> container.

   Bind the form to the data controller that you created in "Step 1: Data Controller" by setting the <form> *controllerId* to match the <dataController> *id* value. For example:

   XML

   ```
   <form controllerId="patientData" id="MyForm" >
   </form>
   ```

   The *id* attribute does not affect the binding but becomes useful in a future step, when we save the form.

2. Within the <form> add two <text> controls.

   Bind each control to a property of the data model by providing a *dataBinding* attribute that identifies a property within the *modelClass* of the <dataController>. Also provide a *label* for each control. For example:

   XML

   ```
   <form controllerId="patientData" id="MyForm" >
     <text label="Patient Name" dataBinding="Name" />
     <text label="Patient City" dataBinding="City" />
   </form>
   ```

   The *label* attribute does not have any purpose relative to the Model View Controller, but it is necessary if we want our controls to have a meaningful labels on the Zen page.

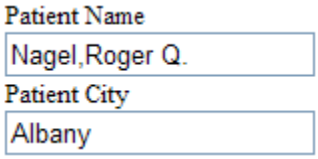
6.5.3 Step 3: Initial Results

View your initial results as follows:

1. Open your Zen page class in Studio.
2. Choose **Build > Compile** or **Ctrl-F7** or the  icon.
3. Choose **View > Web Page** or the  icon.

   The data controller component creates a MyApp.PatientModel object on the server, and asks it to load data from data source record number 1 (identified by the <dataController> *modelId* attribute) into its own properties. The data controller places these data values into the appropriate controls within the form. The *dataBinding* attribute for each control identifies which property provides the value for that control, Name or City.

   The following figure shows our form with the current values from record 1.



6.5.4 Step 4: Saving the Form

Suppose you want the user to be able to edit the values in this form, and to save changes. To save the values in a form associated with a data controller, your application must call the form's **save** method. Typically you would enable this as follows:

1. Add a client-side method to the page class as follows:

   Class Member

   ```
   ClientMethod save() [ Language = javascript ]
   {
     var form = zen('MyForm');
     form.save();
   }
   ```

   Now you can see why it was important to define an *id* for our <form>. The JavaScript function **zen** needs this *id* to get a pointer to the form so that we can call its **save** method.

2. Add to your page a <button> that calls this method to save the form.

   The entire <page> definition now looks something like this:

   XML

   ```
   <page xmlns="http://www.intersystems.com/zen" title="">
     <dataController id="patientData"
                     modelClass="MyApp.PatientModel"
                     modelId="1" />
     <form controllerId="patientData" id="MyForm">
       <text label="Patient Name" dataBinding="Name" />
       <text label="Patient City" dataBinding="City" />
     </form>
     <button caption="Save" onclick="zenPage.save();"/>
   </page>
   ```

3. Try editing the data and clicking **Save**.

Each time the user clicks the **Save** button, the form **save** method calls back to the server and saves the data by calling the appropriate methods of the data model class. During this process, the form asks the data controller to assist with data validation of the various properties. The source of this validation logic is the data model class.

The data controller also has a **save** method we can use. There is a difference between saving the form and saving the controller. Calling the **save** method of the form triggers the form validation logic, after which the form instructs the controller to save data by calling the appropriate methods of the data model class. Saving the controller skips form validation.

You can try out basic form validation in step 3 of this example as follows: If you empty the Patient Name field entirely and click **Save**, a validation error occurs. This is because the Patient property is marked as Required in the ZENDemo.Data.Patient class that serves as our data source. However, if you change the Patient Name or Patient City to any non-empty value, the form saves correctly. It is easier to prove this to yourself once you have extended the form to allow you to easily view more than one data record. Then you can switch back and forth between records to see that they in fact contain your changes.

Note: For further validation examples, try using and viewing the Zen page class ZENMVC.MVCForm in the SAMPLES namespace. Try entering the following URI in the browser:

`http://localhost:57772/csp/samples/ZENMVC.MVCForm.cls`

Where 57772 is the web server port number that you have assigned to Caché. Edit values in one of the forms shown on the page, and click a **Submit** button. You can use Studio to view the class code.

6.5.5 Step 5: Performing Client-side Validation

Rather than wait for server-side validation, we can add client-side validation to the data model class by defining a property-specific **IsValidJS** method. This is a JavaScript `ClientClassMethod` that uses the naming convention *property***IsValidJS** and returns an error message if the value of the given *property* is invalid or `''` (an empty string) if the value is OK. This method can be defined within the data model class, or you can define a datatype class that defines an **IsValidJS** method.

Adding the following method to the MyApp.PatientModel class causes the data controller to automatically apply this validation to the City property on the client each time its **save** method is called:

Class Member

```
ClientClassMethod CityIsValidJS(value) [Language = javascript]
{
  return ('Boston' == value) ? 'Invalid City Name' : '';
}
```

6.5.6 Step 6: Setting Values Programmatically

In order to set data values programmatically, set the value in the controller and then tell the controller to notify all of its views of the change. To set the value, you can use the controller method **setDataByName**, and then use **raiseDataChange** to notify the views. You have to call **raiseDataChange** explicitly, which allows you to change multiple values in the controller and only raise the event once.

Class Member

```
ClientMethod ChangeValue() [ Language = javascript ] {
  var controller = zenPage.getComponentById('patientData');
  controller.setDataByName('Name','Public,John Q');
  controller.raiseDataChange();
}
```

6.6 Adding Behavior to the <form>

The %ZEN.Auxiliary.dataController class offers several useful methods. Suppose you want to enhance your page from the previous exercise, "Binding a <form> to an Object Data Model," so that it can open new records, create new records, delete existent records, or reset the current record to a particular model ID. This topic explain how to accomplish these tasks using dataController methods such as **getModelId** and **setModelId**.

6.6.1 Step 1: Opening a New Record

When you ask the browser to display the page you have been building during these exercises, it always displays the same data record. This is because you have configured the *modelId* property of your <dataController> element with the value of 1. You can see what happens if you change this property to other values, such as 2, 3, or 4, up to 1000.

However, you must remember that these values represent real ID values of existing instances of the ZENDemo.Data.Patient class. These instances exist because all of the classes in ZENDemo.Data are populated automatically when you first run "The Zen Demo" as described in the "Introducing Zen" chapter of *Using Zen*.

Suppose you want to give the user the option of choosing which record to view. There are several options, including:

- Add a text field that allows the user to enter an ID
- Add a combo box that allows the user to choose the record by one of its properties, such as a patient name

- Add a table that allows the user to click on a patient name and see the details on a form below

It does not matter which option you choose for user input. The key task is for your page to be able to tell the data controller to load the record. You can accomplish this as follows:

1. Open your Zen page class in Studio.
2. Add a new text field to the <form>:

   XML

   ```
   <form controllerId="patientData" id="MyForm">
     <text label="ID:"
           onblur="zenPage.loadRecord(zenThis.getValue())"
           dataBinding="%id"/>
     <text label="Patient Name" dataBinding="Name" />
     <text label="Patient City" dataBinding="City" />
   </form>
   ```

3. Add a corresponding client-side method:

   Class Member

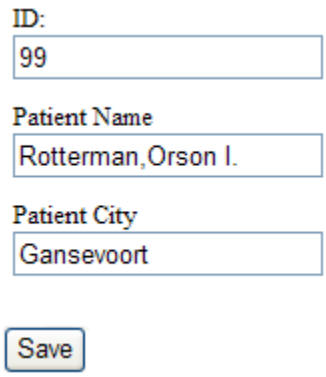
   ```
   ClientMethod loadRecord(id) [ Language = javascript ]
   {
     var controller = zen('patientData');
     controller.setModelId(id);
   }
   ```

   The *onblur* event calls a client-side method **loadRecord** that first gets a pointer to the data controller using its *id* value "patientData", then uses whatever the user has entered in the <text> field as a *modelId* to load the desired record from the data model. To actually load the record, **loadRecord** uses the data controller method **setModelId**.

   Also observe that this example binds the ID field to the `%id` property of the data model, so that this field always shows you the ID of the current record. This step is not necessary for **setModelId** to work, but it is very useful in "Step 2: Creating and Deleting Records" in this exercise.

4. Choose **Build > Compile** or **Ctrl-F7** or the  icon.
5. Choose **View > Web Page** or the  icon.

   The following figure shows the form.

6. Try the *onblur* functionality as follows:
   - Enter 2 (or any other number) in the ID field.
   - Press the **Tab** key to move out of the ID field.
   - The alternate record should display.

7. In the exercise "Binding a <form> to an Object Data Model," during "Step 4: Saving the Form," you added **Save** functionality without the ability to easily test it. Try it now, as follows:
   - Note the number of the current record.
   - Make a significant change to the Name or City field.
   - Click **Save**.
   - Enter a different number in the ID field.
   - Press the **Tab** key to move out of the ID field.
   - The alternate record should display.
   - Enter the number for the record that you saved in the ID field.
   - Press the **Tab** key to move out of the ID field.
   - Your saved changes should be visible on the form.

6.6.2 Step 2: Creating and Deleting Records

Creating and deleting records are also simple tasks. Modify your page to add the necessary buttons and client side code as follows.

1. Open your Zen page class in Studio.
2. After the <form> and before the closing </page>, replace the single **Save** button with the following <hgroup>:

   XML

   ```
   <hgroup>
     <button caption="Save" onclick="zenPage.save()"/>
     <button caption="New" onclick="zenPage.newRecord()"/>
     <button caption="Update" onclick="zenPage.updateRecord()"/>
     <button caption="Delete" onclick="zenPage.deleteRecord()"/>
   </hgroup>
   ```

   These statements add the buttons to an <hgroup> so that they appear on the screen in a row. Each button defines its *onclick* method as a different client-side method.

3. Add the corresponding new client-side methods to the page class:

   Class Member

   ```
   ClientMethod newRecord() [ Language = javascript ]
   {
     var controller = zen('patientData');
     controller.createNewObject();
   }
   ```

   And:

   Class Member

   ```
   ClientMethod updateRecord() [ Language = javascript ]
   {
     var controller = zen('patientData');
     controller.update();
   }
   ```

   And:

Class Member

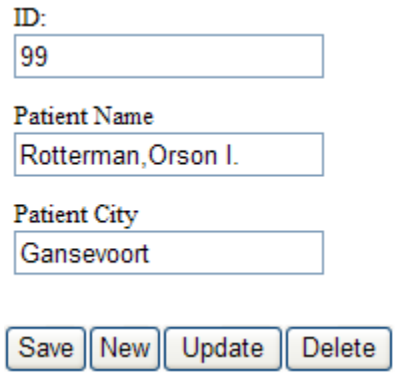
```
ClientMethod deleteRecord() [ Language = javascript ]
{
  var controller = zen('patientData');
  controller.deleteId(controller.getModelId());
  controller.createNewObject();
}
```

Each of these methods uses a different **dataController** method to achieve its purposes.

4. Choose **Build > Compile** or **Ctrl-F7** or the  icon.

5. Choose **View > Web Page** or the  icon.

   The following figure shows the form.



6.6.2.1 New

Suppose the user clicks **New**. Calling **createNewObject** immediately creates a new empty model. In the browser, every time the user clicks **New** the form is emptied. After that, if the user completes the empty form and clicks **Save**, this invokes the form's **save** method. This (eventually) leads to a call to the data model's **%OnSaveSource** method on the server.

In the exercise "Binding a <form> to an Object Data Model," during "Step 4: Saving the Form," you added **Save** functionality by causing the **%OnSaveSource** method to set the `%id` property of the model to the ID of the saved object, as follows:

Class Member

```
Method %OnSaveSource(pSource As ZENDemo.Data.Patient) As %Status
  {
    Set tSC=pSource.%Save()
    If $$$ISOK(tSC) Set ..%id=pSource.%Id()
    Quit tSC
  }
```

As a result, every time the user clicks **New**, enters values, then clicks **Save**, the form shows the newly assigned ID to the user (thanks to the *dataBinding* on that field). Of course, this only works if the data entered in the form passes validation. Name is a required field, so if no Name is entered, the record is not saved.

6.6.2.2 Delete

Suppose the user clicks **Delete**. The data controller's **deleteId** method expects to receive an input argument containing the ID for the record to be deleted. Therefore, when the user clicks **Delete**, the page uses the data controller's **getModelId** method to determine the ID of the record the user is currently viewing. It passes this ID on to **deleteId**. This (eventually) leads to a call to the data model's **%OnDeleteSource** method on the server. The source object is deleted, and since there

is no longer source object, the page calls the data controller's **createNewObject** method to empty the form and prepare it for new input.

Although the code examples in this chapter do not take advantage of this feature, the **deleteId** method returns a Boolean value, true or false. It is true if it successfully deleted the record. It is false if it failed, or if the data controller or its data model are read-only. A data controller is read-only if its *readOnly* attribute is set to 1 (true). A data model is read-only if its class parameter is set to 1 (true).

Important: If, while using this exercise, you delete a record with a specific ID, this object no longer exists. You cannot view or create a record with this ID again.

6.6.2.3 Update

Before clicking **Update**, the user must enter an ID number (between 1 and 1000) in the ID field. The page updates the form fields with data from that record. This fails only if you have previously deleted a record with that ID.

6.6.2.4 Errors

A data controller has a server-side property called *modelError* that is a string containing the most recent error message that the data controller encountered while saving, loading, deleting, or invoking a server-side action. A data controller also has a client-side JavaScript method, **getError**, that an application can invoke to get the *modelError* value. **getError** has no arguments and returns the *modelError* string. It returns an empty string `''` if there is no current error.

6.7 <dynaForm> with an Object Data Model

This topic explains how to use <dynaForm> with a data controller. In this case the data model is an object data model.

6.7.1 Step 1: <dynaForm> is Easy

You may create your first <dynaForm> very easily as follows:

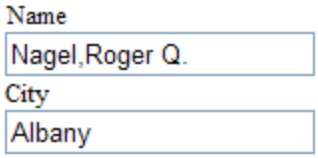
1. Create a new Zen page class. Use the instructions from the exercise "Creating a Zen Page" in the "Zen Tutorial" chapter of *Using Zen*. Call your new class anything you like, but keep it in the MyApp package. Be careful not to overwrite any of your previous work.
2. In XData Contents, place a <dataController> and <dynaForm> inside <page>:

   XML

   ```
   <page xmlns="http://www.intersystems.com/zen" title="">
     <dataController id="patientData"
                     modelClass="MyApp.PatientModel"
                     modelId="1" />
     <dynaForm controllerId="patientData"/>
   </page>
   ```

3. Choose **Build > Compile** or **Ctrl-F7** or the  icon.
4. Choose **View > Web Page** or the  icon.

   The form displays two fields that contain the current Name and City values for the record whose *modelId* you entered in the <dataController> statement. Perhaps you have changed these values, or deleted this record, during previous exercises. Whatever data is now available for that *modelId* displays.



The label for each control is determined by the corresponding property name in MyApp.PatientModel. These labels are different from the text you assigned to the *caption* attribute when you used <form> and <text> components to lay out the form. In all other respects, this display is identical to the display you first saw in the exercise "Binding a <form> to an Object Data Model," during "Step 3: Initial Results." Later steps show how to set specific labels for the controls in a <dynaForm>.

6.7.2 Step 2: Converting to <dynaForm>

Now you are ready to recreate the <form> example from previous exercises in this chapter as a <dynaForm>. To do this, you need to rewrite your MyApp.MyNewPage XData Contents block so that it looks like this:

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">

  <dataController id="patientData"
                  modelClass="MyApp.PatientModel" />

  <dynaForm id="MyForm"
            controllerId="patientData"
            defaultGroupId="generatedFields">
    <text label="ID:"
          onblur="zenPage.loadRecord(zenThis.getValue())"
          dataBinding="%id"/>
    <vgroup id="generatedFields"/>
  </dynaForm>

  <hgroup>
    <button caption="Save" onclick="zenPage.save()"/>
    <button caption="New" onclick="zenPage.newRecord()"/>
    <button caption="Update" onclick="zenPage.updateRecord()"/>
    <button caption="Delete" onclick="zenPage.deleteRecord()"/>
  </hgroup>
</page>
```

That is:

1. In Studio, return to your existing sample page, MyApp.MyNewPage.
2. Remove or comment out this <form>, which specifies each control individually:

   XML

   ```
   <form id="MyForm"
         controllerId="patientData" >
     <text label="ID:"
           onblur="zenPage.loadRecord(zenThis.getValue())"
           dataBinding="%id"/>
     <text label="Patient Name" dataBinding="Name" />
     <text label="Patient City" dataBinding="City" />
   </form>
   ```

3. Add this <dynaForm>, which relies on the data controller to supply it with any control definitions that can be generated based on properties in the data model:

XML

```
<dynaForm id="MyForm"
          controllerId="patientData"
          defaultGroupId="generatedFields">
  <text label="ID:"
        onblur="zenPage.loadRecord(zenThis.getValue())"
        dataBinding="%id"/>
  <vgroup id="generatedFields"/>
</dynaForm>
```

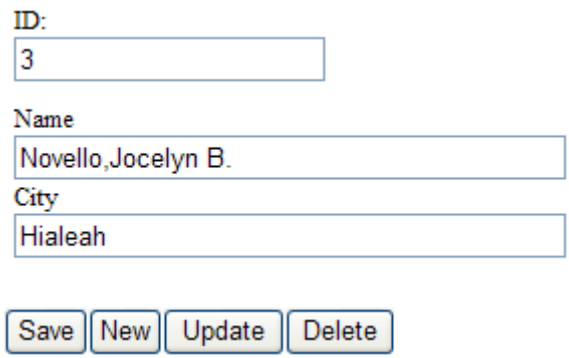
Where:

- *controllerId* identifies the data controller.
- *defaultGroupId* identifies the group, on the form, that contains the controls generated by that data controller.
- *defaultGroupId* is optional; but if it appears in the <dynaGroup> then somewhere inside the <dynaForm>, you must specify a group with an *id* that matches the *defaultGroupId*, in this case a <vgroup>.
- If you want controls to appear on the form that do not depend on the data controller, such as the control whose value is based on the `%id` variable, you must place them explicitly, as shown.

4. Choose **Build > Compile** or **Ctrl-F7** or the  icon.

5. Choose **View > Web Page** or the  icon.

   The following <dynaForm> displays:



6. You may assign specific labels to the generated controls on the <dynaForm> as follows:
   - In Studio, open the object data model class, MyApp.PatientModel.
   - Edit the properties to add the ZENLABEL parameter to each of them:

     Class Member

     ```
     Property Name As %String(ZENLABEL = "Patient Name");
     ```

     And:

     Class Member

     ```
     Property City As %String(ZENLABEL = "Patient City");
     ```

   - Choose **Build > Compile** or **Ctrl-F7** or the  icon to compile the data model.

7. Refresh your Zen page MyApp.MyNewPage in the browser.

   Your <dynaForm> and <form> now produce identical results.

6.7.3 Step 3: Automatic Control Selection

When you use <dynaForm> instead of <form>, it is no longer necessary to add data view components (controls) to the form, item by item, as described in the section "Binding a <form> to an Object Data Model." <dynaForm> automatically extracts this information from the data model at compile time.

The following exercise demonstrates the use of a <dynaForm> by adapting your page class from previous exercises so that it uses a different data model. When you complete the exercise and display the page, a new form appears whose controls are clearly different from those in previous exercises:

1. Return to Studio in the SAMPLES namespace.
2. Choose **Tools > Copy Class**.
3. The Class Copy dialog displays. Enter:
   - **Copy Class** — `MyApp.PatientModel`
   - **To** — `MyApp.EmployeeModel`
   - Select the **Replace Instances of Class Name** check box.
   - Click **OK**.

   The new class definition for MyApp.EmployeeModel displays in Studio.
4. Edit MyApp.EmployeeModel so that it has the following three properties only:

```
Property Name As %String;
Property Salary As %Numeric;
Property Active As %Boolean;
```

5. Edit the methods inside MyApp.EmployeeModel to work with the new properties:

   Class Member

```
Method %OnLoadModel(pSource As ZENDemo.Data.Employee) As %Status
{
  Set ..Name = pSource.Name
  Set ..Salary = pSource.Salary
  Set ..Active = pSource.Active
  Quit $$$OK
}
```

   And:

   Class Member

```
Method %OnStoreModel(pSource As ZENDemo.Data.Employee) As %Status
{
  Set pSource.Name = ..Name
  Set pSource.Salary = ..Salary
  Set pSource.Active = ..Active
  Quit $$$OK
}
```

6. Choose **Build > Compile** or **Ctrl-F7** or the  icon.
7. Choose **Tools > Copy Class**.
8. The Class Copy dialog displays. Enter:
   - **Copy Class** — `MyApp.MyNewPage`
   - **To** — `MyApp.MyOtherPage`

- Select the **Replace Instances of Class Name** check box.
- Click **OK**.

The new class definition for MyApp.MyOtherPage displays in Studio.

9. Take a shortcut by leaving the <dataController> *id* as "patientData".

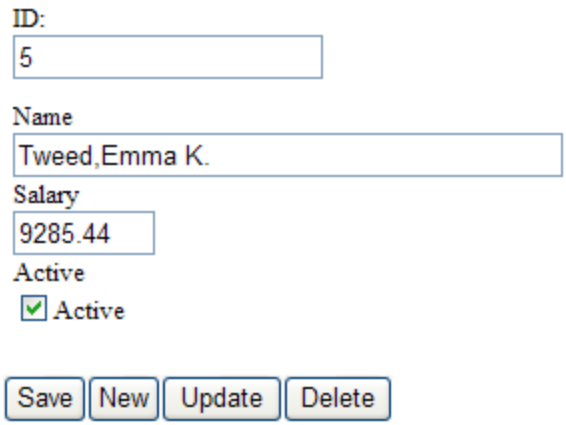
   You may ignore this shortcut by globally replacing "patientData" with a more meaningful *id*, for example "employeeData". However, make sure you change all the instances of this *id* string in the class to avoid errors at runtime.

10. Change the <dataController> *modelClass* to "MyApp.EmployeeModel".

11. Choose **Build > Compile** or **Ctrl-F7** or the  icon.

12. Choose **View > Web Page** or the  icon.

    The following figure shows the resulting form, with the values for record 5.



<dynaForm> has chosen controls for this form as follows:

- <text> for the Name, which is a %String
- <text> for the Salary, which is %Numeric
- <checkbox> for the Active status, which is a %Boolean

6.7.3.1 <dynaForm> Controls Based on Data Types

<dynaForm> determines which type of control to assign to each property in the model based on the data type of that property. The following table match property data types with the <dynaForm> controls they generate.

Note: If you do not like the choices that <dynaForm> makes, you can switch to <form> and bind each control to a property individually, using the *dataBinding* attribute as described in the exercise "Binding a <form> to an Object Data Model" during "Step 2: Data View."

Table 6–1: <dynaForm> Controls Based on Data Types

Data Type	Details	Control
%ArrayOfDataTypes	See "Array Data Types" following the table.	<textarea>
%Boolean	—	<checkbox>
%Date	In YYYY-MM-DD format	<dateSelect>
%Date	In other formats	<text>
%Enumerated	Using a VALUELIST with 4 or fewer values.	<radioSet>
%Enumerated	Using a VALUELIST with more than 4 values	<combobox>
%ListOfDataTypes	See "List Data Types" following the table.	<textarea>
%Numeric	—	<text>
Object reference	<dynaForm> generates an SQL query	<dataCombo>
Stream	%CharacterStream	<textarea>
Stream	%BinaryStream	<image>
%String	With MAXLEN over 250	<textarea>
%String	With MAXLEN between 1 and 250	<text>
Public properties	All types not listed above	<text>
Private properties	Any properties marked private	Not displayed

Most of the data types listed in the previous table are defined as Caché classes. As such, they can define class parameters, including the VALUELIST, DISPLAYLIST, and MAXLEN parameters mentioned in the table. These parameters provide details about the data type.

For %Enumerated properties, the VALUELIST parameter specifies the internal values (1, 2, 3) and the DISPLAYLIST parameter specifies the names that are displayed for the user to choose (High, Medium, Low). For %String properties, the MAXLEN parameter specifies a maximum length.

Many other class parameters are available. For details, see the "Parameters" section in the "Data Types" chapter of *Using Caché Objects*.

List Data Types

For a property whose data type is %ListOfDataTypes, Zen streams the list collection to the client as one string delimited by carriage return characters. The resulting <textarea> control displays one collection item per line of text.

For a <dynaForm>, this convention works when the data model class sets this property parameter:

```
ZENCONTROL="textarea"
```

For details, see "Data Model Property Parameters" in the "Data Model Classes" section of this chapter.

For a <form>, this convention works when you bind a <textarea> control to the property of type %ListOfDataTypes using the *dataBinding* attribute.

See the exercise "Binding a <form> to an Object Data Model" during "Step 2: Data View."

Array Data Types

For a property whose data type is %ArrayOfDataTypes, the conventions are the same as for %ListOfDataTypes, except that the serialized string takes this form:

key:*value*[CR]*key*:*value*[CR]*key*:*value*

Where : is a single colon and [CR] represents a single carriage return character.

6.8 <dynaForm> with an Adaptor Data Model

This topic explains how to use <dynaForm> with a data controller when the data model is an adaptor data model. This approach is particularly convenient when you have an existing class with a large number of properties that you need to display on a form. In that case it would be extremely time-consuming to add these properties one by one to a subclass of %ZEN.DataModel.ObjectDataModel, as demonstrated in the previous exercises in this chapter.

<dynaForm> can save coding time, especially when you use it in combination with a subclass of %ZEN.DataModel.Adaptor. All you need to do then is to create a Zen page class whose <page> contains a <dataController> and a <dynaForm>. Your subclass of %ZEN.DataModel.Adaptor becomes the model, the view, *and* the controller, all in one. All of its properties become controls on the resulting <dynaForm>. Zen generates the appropriate control type for property automatically.

6.8.1 Step 1: Generating the Form

The basic outline for using <dynaForm> with an adaptor data model is as follows:

Note: This example is based on the Zen classes ZENMVC.MVCDynaForm, ZENMVC.Person, and ZENMVC.Address in the SAMPLES namespace.

1. Edit a persistent class so that it also extends %ZEN.DataModel.Adaptor, for example:

   Class Definition

   ```
   Class ZENMVC.Person Extends (%Persistent, %ZEN.DataModel.Adaptor)
   {
     Property Name As %String [ Required ];
     Property SSN As %String;
     Property DOB As %Date;
     Property Salary As %Numeric;
     Property Active As %Boolean;
     Property Home As Address;
     Property Business As Address;
   }
   ```

   The Person class includes properties defined by another class, Address, which must also extend %ZEN.DataModel.Adaptor but which need not be persistent, for example:

   Class Definition

   ```
   Class ZENMVC.Address Extends (%SerialObject, %ZEN.DataModel.Adaptor)
   {
     Property City As %String(MAXLEN = 50);
     Property State As %String(MAXLEN = 2);
     Property Zip As %String(MAXLEN = 15);
   }
   ```

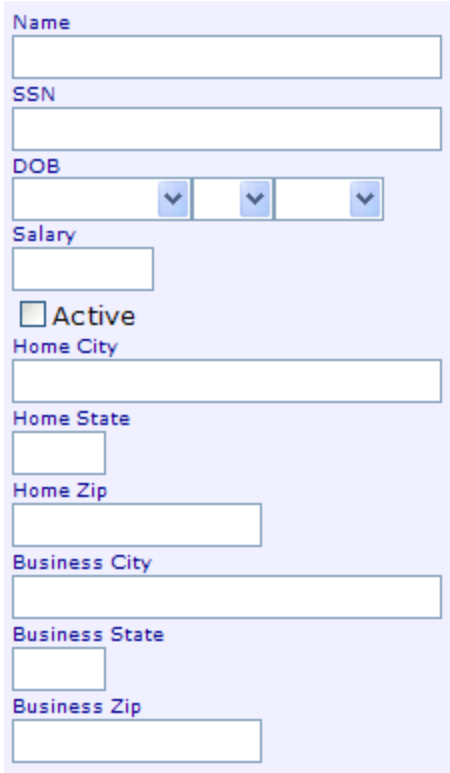
2. Compile both data model classes.
3. Create a new Zen page class.
4. In XData Contents, place a <dataController> and <dynaForm> inside <page>:

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">
  <dataController id="source" modelClass="ZENMVC.Person" modelId=""/>
  <dynaForm id="MyForm" controllerId="source" />
</page>
```

5. Compile the Zen page class.
6. Choose **View > Web Page** or the  icon.

   <dynaForm> generates the appropriate controls and displays the form, for example:



For a table that matches data types with the <dynaForm> controls they generate, see the "<dynaForm> Controls Based on Data Types" table in this chapter.

6.8.2 Step 2: Property Parameters

Once you have the basics in place, you may refine the form by assigning data model parameters to the properties in the persistent class that you are using as an adaptor data model. For example:

1. Open the data model class Person from "Step 1: Generating the Form" in this exercise.
2. Add data model parameters to the properties as shown below:

Class Definition

```
Class ZENMVC.Person Extends (%Persistent, %ZEN.DataModel.Adaptor)
{
  Property Name As %String (ZENLABEL = "Employee Name") [ Required ];
  Property SSN As %String (ZENREADONLY = 1);
  Property DOB As %Date (ZENLABEL = "DateofBirth", ZENATTRS="format:DMY");
  Property Salary As %Numeric (ZENHIDDEN = 1);
  Property Active As %Boolean
    (ZENLABEL = "Is this person working?", ZENATTRS="showLabel:false");
  Property Home As Address;
  Property Business As Address;
}
```

3. Compile the Person class.
4. Refresh your view of the Zen page class in the browser.

   <dynaForm> generates the appropriate controls and displays the form, for example:

   The data model parameters have the following effects:

   - ZENATTRS applies the specified attribute(s) and value(s) to the generated control.
   - ZENHIDDEN hides the control associated with that property.
   - ZENLABEL replaces the default label (the property name) with a custom string.
   - ZENREADONLY displays the control but prevents the user from editing its contents.

For a table that lists more data model parameters, and explains their effects on generated controls in the <dynaForm>, see the section "Data Model Property Parameters."

6.8.3 Step 3: Adding Behavior to the <dynaForm>

Adding behavior to a <dynaForm> is quite similar to the steps for <form>. For <dynaForm>, the steps are:

1. Open the Zen page class from "Step 2: Property Parameters."

2. Add a <text> control to display the current model ID value, and buttons to permit Update, New, and Save operations, as follows:

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">

  <text label="Model ID:" id="idText" dataBinding="%id"
        onblur="zenPage.loadRecord(zenThis.getValue())" />
  <spacer height="10"/>

  <dataController id="source" modelClass="ZENMVC.Person" modelId=""/>
  <dynaForm id="MyForm" controllerId="source" />

  <hgroup>
    <button caption="Update" onclick="zenPage.showRecord();" />
    <button caption="New" onclick="zenPage.newRecord();" />
    <button caption="Save" onclick="zenPage.save();" />
  </hgroup>

</page>
```

3. Add the corresponding new client-side methods to the page class:

Class Member

```
ClientMethod loadRecord(id) [ Language = javascript ]
{
  var controller = zen('source');
  controller.setModelId(id);
}
```

And:

Class Member

```
ClientMethod showRecord() [ Language = javascript ]
{
  var controller = zen('source');
  controller.update();
}
```

And:

Class Member

```
ClientMethod newRecord() [ Language = javascript ]
{
  var text = zen('idText');
  text.setValue("");
  var controller = zen('source');
  controller.createNewObject();
}
```

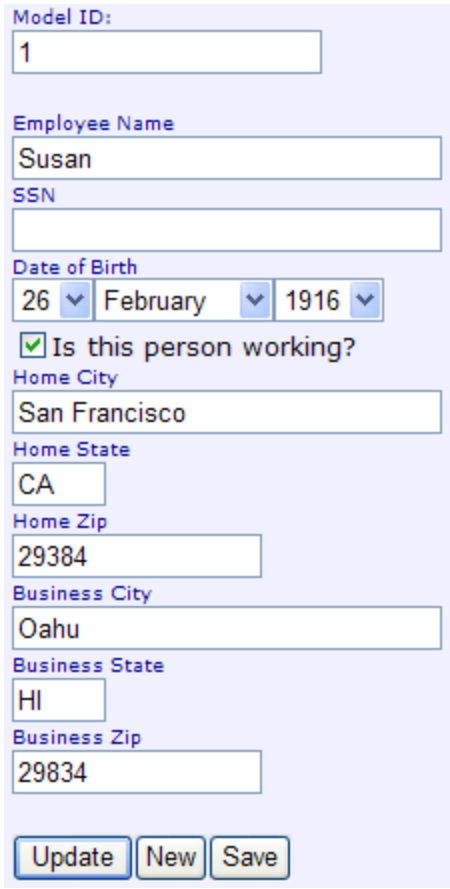
And:

Class Member

```
ClientMethod save() [ Language = javascript ]
{
  var form = zen('MyForm');
  form.save();
}
```

4. Compile the Zen page class.

5. Choose **View > Web Page** or the  icon.
6. Click the **New** button.
7. Enter data in the fields (except the Model ID and the read-only SSN field) and click **Save**.
8. Click the **New** button again, just to clear the fields.
9. Enter the number 1 in the ID field and click Update. The corresponding record displays. For example:



Since you have provided no special format for model ID values in the Person class, the default prevails. This means each new record you add gets a sequential number starting at 1. You may add more records by repeating steps 6 and 7. The numbers increment automatically.

If you try to view a record by entering an ID number that does not exist, the <dynaForm> displays with all of its fields disabled. You may click New to redisplay an active form in which to enter data for a new record.

6.8.4 Step 4: Virtual Properties

If you want to interject changes in a data model before using it, you can override the **%OnGetPropertyInfo** method in the data model class. This is the way to add *virtual properties* that you want to use in the model, but that do not exist in the class that you began with. In order to use virtual properties, you must ensure that the data model class parameter DYNAMICPROPERTIES is set to 1 (true). Its default value in the %ZEN.DataModel.Adaptor class is 0 (false). You can use **%OnGetPropertyInfo** with either type of data model, but it makes the most sense for an adaptor data model because in that case you are using an existing class as a data model.

This exercise adds a **%OnGetPropertyInfo** method to the adaptor data model class Person from the previous exercises in this chapter:

1. Open the Person class in Studio.
2. Choose **Class > Override**.
3. Select **%OnGetPropertyInfo** and click **OK**.

   Studio adds a skeleton **%OnGetPropertyInfo** method to the class.
4. Edit the method as follows:

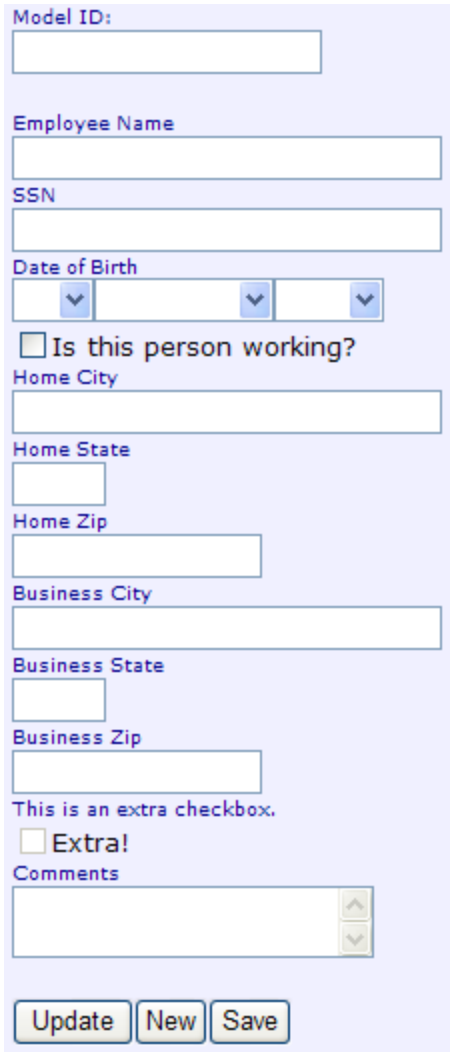
   These statements add a <checkbox> and a <textarea> to any <dynaForm> generated by the model.

   ### Class Member

```
ClassMethod %OnGetPropertyInfo(pIndex As %Integer,
                               ByRef pInfo As %String,
                               pExtended As %Boolean = 0) As %Status
{
  #; Increment past the 3 embedded properties from the last Address object.
  #; This is not necessary when the last property in the Person object
  #; is a simple data type such as %String or %Boolean or %Numeric.
  Set pIndex = pIndex + 3

  #; add a field at the end of the form
  Set pInfo("Extra") = pIndex
  Set pInfo("Extra","%type") = "checkbox"
  Set pInfo("Extra","caption") = "Extra!"
  Set pInfo("Extra","label") = "This is an extra checkbox."
  Set pIndex = pIndex + 1

  #; add another field at the end of the form
  Set pInfo("Comments") = pIndex
  Set pInfo("Comments","%type") = "textarea"
  Set pInfo("Comments","caption") = "Please enter additional comments:"
   Set pIndex = pIndex + 1

  Quit $$$OK
}
```

5. Recompile the Person class.
6. Refresh your view of the Zen page class in the browser.

   <dynaForm> generates the appropriate controls and displays the form, for example:



6.9 Data Model Classes

The basic behavior of data models comes from the abstract base class %ZEN.DataModel.DataModel. This class defines the basic data model interface which is, in turn, implemented by subclasses. A data model class can be one of the following types:

- The data model class serves as a wrapper for an independent data source. The data model object provides the interface and the source object (or objects) provide the actual data. In this case the data model class must implement the additional callback methods related to the data source object. Our form examples followed this convention by subclassing %ZEN.DataModel.ObjectDataModel and overriding several server-side callback methods.
- The data model class is also the data source object. The data model object provides both the data and the interface. In this case, there is no need to override the callback methods related to the data source object. All you need to do is to make your data source class implement the %ZEN.DataModel.Adaptor interface. Then you can use the resulting class directly as the *modelClass* for the <dataController> component in your page class.

6.9.1 Data Model Class Properties

The %ZEN.DataModel.DataModel class provides the following properties:

- String that identifies the currently active instance of the data model object, also known as the model ID. This value can be initially set by providing a *modelId* attribute in the <dataController> definition. The exact form and possible values of the model ID are up to the developer of a specific data model class. The property names are:
  - `dataModelId` — Use in JavaScript code that runs on the client side
  - `%id` — Use in ObjectScript, Caché Basic, or Caché MVBasic code that runs on the server side

  For examples, see "Adding Behavior to the <form>" in this chapter.
- Array of strings that provide display names for the data series in the model. Each string labels one data series. The array is subscripted by series number (1–based). The property names are:
  - `seriesNames` — Use in JavaScript code that runs on the client side
  - `%seriesNames` — Use in ObjectScript, Caché Basic, or Caché MVBasic code that runs on the server side
- Number of data series contained within the data model. The property names are:
  - `seriesCount` — Use in JavaScript code that runs on the client side
  - `%seriesCount` — Use in ObjectScript, Caché Basic, or Caché MVBasic code that runs on the server side

  For details, see the section "Data Model Series."

6.9.2 Data Model Class Parameters

Data model classes provide class parameters that determine the type of data model. The following table lists them.

Table 6–2: Data Model Class Parameters

Property Parameter	Description
DOMAIN	Available for subclasses of %ZEN.DataModel.ObjectDataModel only. You must provide a value for this parameter if you wish to use Zen localization.
DYNAMICPROPERTIES	1 (true) or 0 (false). If true, this model supports virtual properties. For background information, see the section "Virtual Properties." The default value for DYNAMICPROPERTIES is: • 1 (true) for %ZEN.DataModel.ObjectDataModel • 0 (false) for %ZEN.DataModel.Adaptor
READONLYMODEL	1 (true) or 0 (false). If true, indicates that this is a read-only model. It can be used to display data but not to generate editable forms. The default is 0 (false).

6.9.3 Data Model Property Parameters

The %ZEN.DataModel.ObjectDataModel class provides property parameters that you can apply to the data model properties that you wish to use as controls on a form. These parameters let you provide more specific control over the properties of

the data model class. The utility class %ZEN.DataModel.objectModelParameters defines these parameters; the following table lists them.

Note: For examples, use Studio to view the classes ZENMVC.FormDataModel and ZENMVC.FormDataModel2 in the SAMPLES namespace. Also see the exercise "<dynaForm> with an Adaptor Data Model" in this chapter.

Table 6–3: Data Model Property Parameters

Property Parameter	Description
ZENATTRS	List of additional attributes to apply to the control used for this property. This string should have the following form: `"attribute:value|attribute:value"`

ZENCONTROL	Type of control used to display this property within a form; If not defined, Zen chooses the control type based on the data type of the property.
	If you specify a simple class name as the value of ZENCONTROL, Zen assumes that this class is in the package %ZEN.Component. The following example specifies the class %ZEN.Component.textarea:
	`ZENCONTROL = "textarea"`
	You can also specify a full package and class name as the value of ZENCONTROL. The package and class must reside in the same namespace as the class that is defining the ZENCONTROL parameter value. The following example specifies a custom component class:
	`ZENCONTROL = "MyPackage.Utils.textAreaAppendable"`

ZENDISPLAYCOLUMN	If defined, this is the name of the column used to provide a display value for SQL statements automatically generated for this property.
ZENGROUP	The *id* of a group component that the control used for this property should be added to. This provides a way to control layout. If not defined, the control is added directly to the form.
ZENHIDDEN	1 (true) or 0 (false). If true, indicates that this is a hidden field. When the value of this field is sent to the client, it is not displayed. The default is 0 (false).
ZENLABEL	Label used for this property within a form. The label text cannot contain the comma (,) character, because it is added to a comma-delimited list of labels.
ZENREADONLY	1 (true) or 0 (false). If true, this is a read-only field and cannot be edited by the user. The default for is 0 (false).
ZENSIZE	The ZENSIZE parameter provides a value for the size property of a control, if the control has one. The interpretation of size depends on the HTML element created by the control. For example, for a text control, size is proportional to the number of characters displayed. This behavior is defined by HTML, not Zen.
ZENSQL	If defined, this is an SQL statement used to find possible values for this property. This parameter corresponds to the *sql* property of the various data-driven Zen components. For details, see the "Specifying an SQL Query" section in the chapter "Zen Tables."

Property Parameter	Description
ZENSQLLOOKUP	If defined, this is an SQL statement used to find the appropriate display value for a given logical value. This parameter corresponds to the *sqlLookup* property of data-driven Zen components like <dataListBox> and <dataCombo>. For details, see the "<dataCombo> Logical and Display Values" section in the chapter "Zen Controls."
ZENTAB	A positive integer. If specified, this overrides the (1–based) default tab order of the control used to display the property within a form. All controls with ZENTAB specified are placed before controls that do not define it.
ZENTITLE	Optional popup title string displayed for this property within a form.

6.9.4 Value Lists and Display Lists

Some of the fields in a form associated with a data model might need separate value lists and display lists. Both are lists of strings. The value list gives the logical values for storage on the server, and the display list specifies the choices that the application displays to the user on the client. These concepts apply to an MVC data model as follows:

- If you are using a <form> with a data model, this concept applies to any of the controls that offer *valueList* and *displayList* attributes: <radioSet>, <select>, or <combobox>. For details, see the chapter "Zen Controls."
- If you are using a <dynaForm> with a data model, this concept applies to any property whose data type uses VALUELIST and DISPLAYLIST class parameters, for example a property of type %Enumerated. For a table that matches data types with the <dynaForm> controls they generate, see the "<dynaForm> Controls Based on Data Types" table earlier in this chapter.

These controls (or in the case of <dynaForm>, the controls that Zen automatically generates to represent these properties) show their display lists on the client. The data model always converts values to the display format before sending them to the client. If the Zen application is *not* localized, the client-side value list and display list are both the same: they are identical to the server-side display list. Any client-side logic for this control must expect these values.

If the Zen application is localized into multiple languages, and if the data model class correctly defines the DOMAIN class parameter, then the conventions are a bit different. The client-side value list is still the same as the server-side display list, but now the client-side display list consists of the server-side display values in the local language. Any client-side logic for this control must expect these values.

As an example, suppose a property in an MVC data model class uses VALUELIST and DISPLAYLIST as follows:

Class Member

```
Property Sex As %String(VALUELIST=",1,2", DISPLAYLIST=",Male,Female");
```

In this case, the logical value of Sex is 1 or 2. This is what is stored in the database and this is what server-side logic uses. An MVC form only sees the display values. Specifically it sees something like this:

```
radioSet.valueList = "Male,Female"
radioSet.displayList = $$$Text("Male,Female")
```

Note: For more about localization, the DOMAIN parameter, and $$$Text macros, see the "Zen Localization" chapter in *Developing Zen Applications*.

6.9.5 Object Data Model Callback Methods

When you create an object data model, you subclass %ZEN.DataModel.ObjectDataModel and provide implementations for its server-side callback methods. The following table describes these methods in detail. You first encountered several of these methods in the exercise "Constructing a Model" during "Step 2: Object Data Model". For these methods the **Example** column contains the word "Yes."

Table 6–4: Object Data Model Callback Methods

Method	Example	This Callback Method is Invoked When...
%OnDeleteModel	—	The data model is deleted. This method is implemented by the subclasses of the data model class, if they exist.
%OnDeleteSource	Yes	The data model is deleted. If implemented, it is responsible for deleting the object that has the given id and returning the status code resulting from that operation.
%OnGetPropertyInfo	—	The **%GetPropertyInfo** method invokes it. See the discussion following this table.
%OnInvokeAction	—	A user-defined, named action is invoked on this model object. See the discussion following this table. This method is implemented by the subclasses of the data model class, if they exist.
%OnLoadModel	Yes	Zen does the actual work of loading values from the data source into the data model object. The only data to load is the data that is actually seen by the user. This is the place to perform any aggregation or other operations on the data before storing it.
%OnNewSource	Yes	A data model needs a new instance. If implemented, it opens a new (unsaved) instance of the data source object used by the data model, and return its reference.
%OnOpenSource	Yes	A data model is opened. If implemented, it opens an instance of the data source object used by the data model, and returns its reference
%OnSaveSource	Yes	The data model is saved. If implemented, it is responsible for saving changes to the data source. It saves the given source object and return the status code resulting from that operation. Before returning the status code, it sets the data model's `%id` property to the identifier for the source object.
%OnStoreModel	Yes	Zen does the actual work of copying values from the data model to the data source. This method loads data from the model (probably changed by the user through a form) back into the source object.
%OnSubmit	—	A form connected to this data model is submitted. The contents of this data model are filled in from the submitted values before this callback is invoked. Implementing this callback is optional.

6.9.6 Virtual Properties

When a data controller needs to find information about the properties within a data model, it calls the data model's **%GetPropertyInfo** method. This returns a multidimensional array containing details about the properties of the data model. The code that assembles this information is automatically generated based on the properties, property types, and property parameters of the data model class.

A data model class can modify the property information returned by **%GetPropertyInfo** by overriding the **%OnGetPropertyInfo** callback method. **%GetPropertyInfo** invokes the **%OnGetPropertyInfo** immediately before it returns the property information. **%OnGetPropertyInfo** receives, by reference, the multidimensional array containing the property information. **%OnGetPropertyInfo** can modify the contents of this array as it sees fit. Properties can be added, removed, or have their attributes changed. Attributes that you add using this method are called *virtual properties*. In order to use virtual properties, you must ensure that the data model class parameter DYNAMICPROPERTIES is set to 1 (true).

Note: For examples, see the exercise "<dynaForm> with an Adaptor Data Model" during "Step 4: Virtual Properties."

The **%OnGetPropertyInfo** signature looks like this:

```
ClassMethod %OnGetPropertyInfo(pIndex As %Integer,
                               ByRef pInfo As %String,
                               pExtended As %Boolean = 0,
                               pModelId As %String = "",
                               pContainer As %String = "") As %Status
```

Where:

- *pIndex* is the index number that should be used to add the next property to the list.
- *pInfo* is a multidimensional array containing information about the properties of this data model.
- If *pExtended* is true, then complete information about the properties should be returned; if false, then only property names need be returned (applications can simply ignore this).
- *pModelId* a string that identifies the currently active instance of the data model object, also known as the model ID. This is provided for cases where the contents of a dynamic form may vary by instance of the data model object. The exact form and possible values of the model ID are up to the developer of a specific data model class.
- If this is an embedded property, *pContainer* is the name of the property that contains it.

Within the **%OnGetPropertyInfo** method, the property information array `pInfo` is subscripted by property name. The top node for each property contains an integer index number used to determine the ordinal position of a property within a dynamically generated form:

If you want **%OnGetPropertyInfo** to add a new property to a data model, simply add the appropriate nodes to the property information array. The new property is treated as a "virtual" property. That is, you can set and get its value by name even though there is no property formally defined with this name. When adding a new property in **%OnGetPropertyInfo**, set the top level node to the current index number and then increment the index by 1:

```
pInfo("Property") = pIndex
Set pIndex = pIndex + 1
```

The property information array has a number of subnodes that can be defined to provide values for other property attributes. Built-in attributes start with % and include:

- `pInfo("Property","%type") = "control"`

  The `%type` subnode identifies the name of the Zen control that should be used for properties of this type when using a dynamic form. Note that this name is the class name of a component. If no package name is provided, it is assumed that this is a component in the %ZEN.Component package. Use a full class name if you wish to specify a component from a different package.

- `pInfo("Property","%group") = "groupId"`

  The `%group` subnode indicates the *id* of a group component contained by a dynamic form. If `%group` is specified and there is a group with this *id*, then the control for this property is created within this group. This provides a way to control the layout of controls within a dynamic form.

Attributes that do not start with a % specify values that should be applied to a property of the control with the same name. For example, the following statement causes a dynamic form to set the *label* property of the control used for the MyProp property to "My Label".

```
Set pInfo("MyProp","label") = "This is an extra field!"
```

Data models and data controllers each support the **%OnGetPropertyInfo** method. At runtime, the order in which Zen adds controls to the generated form is as follows:

1. Creates an initial list of controls based on the data model properties and their parameters.
2. Modifies this list of controls by calling the data model's **%OnGetPropertyInfo** method, if present.
3. Further modifies this list of controls by calling the data controller's **%OnGetPropertyInfo** method, if present.

6.9.7 Controller Actions

The **%OnInvokeAction** callback lets you define "actions" that can be invoked on the data model via the data controller. The client can invoke an action by calling the dataController's **invokeAction** method as follows:

```
controller.invokeAction('MyAction',data);
```

This, in turn, invokes the server-side **%OnInvokeAction** callback of the data model, passing it the name of the action and the data value. The interpretation of the action name and data is up to the application developer.

6.9.8 Data Model Series

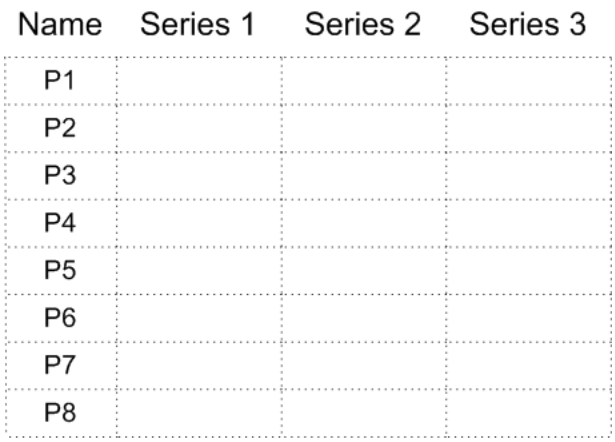
The basic data model object consists of a series of name-value pairs.

Figure 6–4: Data Model with Name-Value Pairs

Name	Value
P1	
P2	
P3	
P4	
P5	
P6	
P7	
P8	

The name-value pairs in the data model comprise all of the properties in the data model class, minus those properties marked ZENHIDDEN, plus any properties added by **%OnGetPropertyInfo**, minus any properties deleted by **%OnGetProperty**. By default, the number of series is 1, but it could be larger. If there are multiple series in the model, conceptually it becomes a matrix.

Figure 6–5: Data Model with Data Series



Name	Series 1	Series 2	Series 3
P1			
P2			
P3			
P4			
P5			
P6			
P7			
P8			

You can add multiple series to the model if you write your own %OnLoadModel method, as in the SAMPLES class ZENMVC.ChartDataModel2, shown below. This example creates three series for the model and assigns values to data model properties in each of the series.

Class Definition

```
Class ZENMVC.ChartDataModel2 Extends %ZEN.DataModel.ObjectDataModel
{
Property Cars As %Integer;
Property Trucks As %Integer;
Property Trains As %Integer;
Property Airplanes As %Integer;
Property Ships As %Integer;

Method %OnLoadModel(pSource As %RegisteredObject) As %Status
{
  Set scale = 100

  #; This model has multiple data series. We set up the data series here.
  Set ..%seriesCount = 3
  Set ..%seriesNames(1) = "USA"
  Set ..%seriesNames(2) = "Europe"
  Set ..%seriesNames(3) = "Asia"

  #; Now we provide data for each property within each series.
  #; We use the %data array so that we can address multiple series.
  For n = 1:1:..%seriesCount {
    Set ..%data(n,"Cars") = $RANDOM(100) * scale
    Set ..%data(n,"Trucks") = $RANDOM(100) * scale
    Set ..%data(n,"Trains") = $RANDOM(100) * scale
    Set ..%data(n,"Airplanes") = $RANDOM(100) * scale
    Set ..%data(n,"Ships") = $RANDOM(100) * scale
    }
  Quit $$$OK
  }
}
```

When your data model has multiple series, if you bind the data model to a chart, the chart automatically picks up the various series, although you need to be careful with a pie chart. Series work similarly for a grid. A form can only display one series at a time, so you need to rely on the data controller attribute *defaultSeries* to determine which series is currently in view.

6.9.9 Custom Data Model Classes

%ZEN.DataModel.ObjectDataModel or %ZEN.DataModel.Adaptor are sufficient for most needs. However, sometimes a developer might want to create a special category of data model, for example to represent a global. In that case the developer must subclass %ZEN.DataModel.DataModel and implement the details of this subclass.

The following table lists methods that applications can call in order to work with a data model object. The behavior of these methods is up to the specific %ZEN.DataModel.DataModel subclass that implements them.

Table 6–5: Custom Data Model Class Methods

Method	Description
%DeleteModel	Delete an instance of a data model object given an identifier value. This takes the given identifier value and uses it to delete an instance of a source object (if applicable). (If the data model object serves as both interface and data source, then the data model object itself is deleted).
%OpenModel	Open an instance of a data model object given an identifier value. This takes the given identifier value and uses it to find an instance of a source object (if applicable) and then copies the appropriate values of the source object into the properties of the data model object. (If the data model object serves as both interface and data source, then this copying is not carried out).
%SaveModel	Save an instance of a data model object. This copies the properties of the data model object back to the appropriate source object (if applicable) and then asks the source object to save itself. (If the data model object serves as both interface and data source, then the data model itself is saved.).

7

Navigation Components

The "Zen Layout" chapter of *Using Zen* introduces Zen group components as the key to laying out the Zen page. In that chapter, the "Groups" section describes the simple group components <hgroup>, <vgroup>, <page>, <pane>, and <spacer>.

This chapter describes a more complex set of group and menu components. A developer can use these components to support navigation through Zen applications:

- "Links" link to other Zen pages, or other application content, via a URI.
- "Menus" present and manage a structured set of choices, usually links.
- "Navigator" creates a navigation interface similar to that found on mobile devices.
- "Tabs" define tabbed menus and tabbed forms.
- "Trees" provide a hierarchical outline of links that expands or contracts in response to user clicks.
- "Filters" allow you to filter the available choices by category.

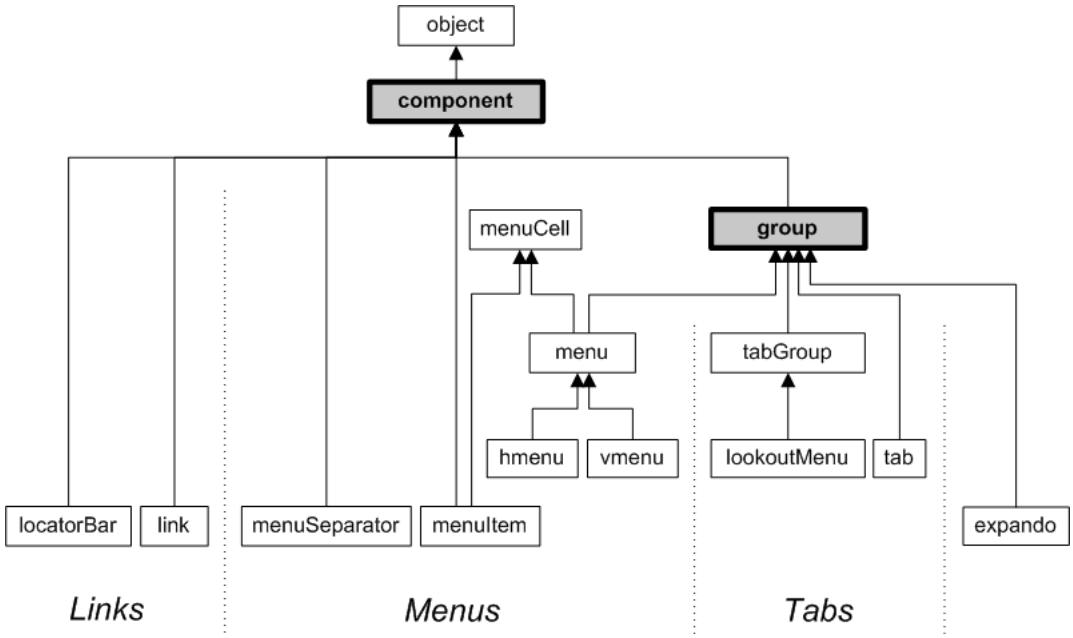
The following table organizes Zen navigation components into two categories: containers (at left) and contained (at right). This chapter describes each component in the order listed in the table.

Container	Purpose	Components It Typically Contains
(Any)	Provide a link to another page or to a popup message.	<link> for each link.
<locatorBar>	Navigation bar for the top of the page.	<locatorLink> for each link in the navigation sequence.
<menu>	Menu container. The default layout is vertical but you can use the *layout* attribute to change to horizontal layout. Alternatively, you may use <hmenu> or <vmenu>.	<menuItem> for each option. <menuSeparator> for the space between options.
<hmenu>	Horizontally oriented menu	
<vmenu>	Vertically oriented menu	

Container	Purpose	Components It Typically Contains
<tabGroup>	Traditional set of tabs from which the user can choose one to be displayed on top of the set.	<tab> for each option.
<lookoutMenu>	A tabbed menu that displays a button for each tab. Clicking on a button makes the tab contents display beneath the button. Each tab consists of menu items.	
<tab>	A tab is a group that may contain any combination of components.	Inside a <lookoutMenu>, each <tab> contains <menuItem> and <menuSeparator> components.
<expando>	Expanding and contracting group which may contain any combination of components.	For a simple navigation tree, you can provide <link> components within <expando>.
<dynaTree>	Expandable tree of links.	Generated links. These either use the data within the global or are supplied by the callback. <dynaTree> is not a group and does not contain other components.
<buttonView>	Set of links that permits filtering according to categories.	Buttons laid out in a grid according to the number of columns specified. <buttonView> is not a group and does not contain other components.

The following figure lists most of the components described in this chapter. All of the classes shown in the diagram are in the package %ZEN.Component, for example %ZEN.Component.tab. The diagram shows the inheritance relationships for these classes. It also highlights which of these components can contain other components, by showing which components inherit from %ZEN.Component.group.

Figure 7–1: Zen Navigation Components



7.1 Links

The following Zen components provide links to content that is available via a URI:

- "<link>"
- "<locatorBar>"

7.1.1 <link>

The <link> component outputs a simple link (an HTML anchor element) to the Zen page. The default formatting for this type of link is to use color and underline it. <link> has the following attributes:

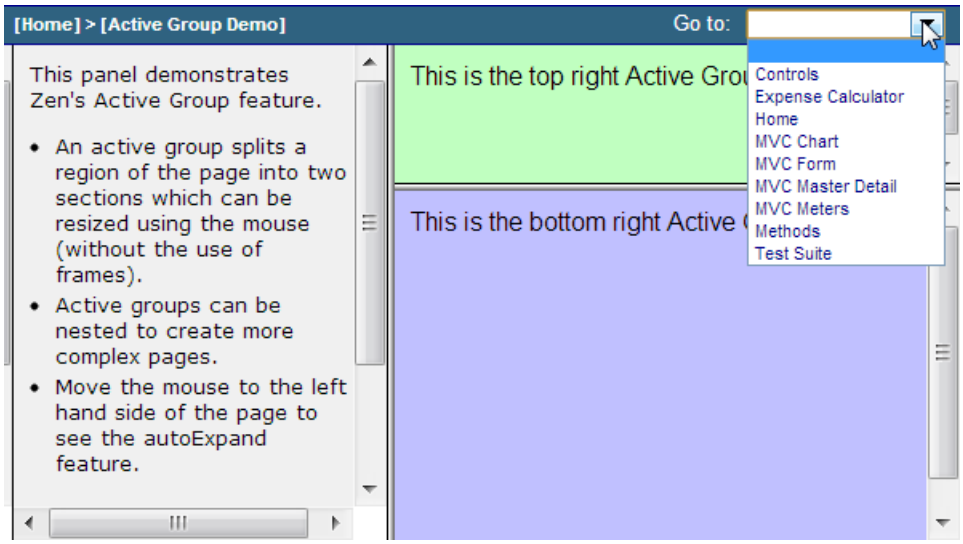
Attribute	Description
Zen component attributes	<link> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
caption	Text for this link in the Zen page display. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types." The *caption* value can be a literal string, or it can contain a Zen #()# runtime expression.
disabled	If true, this link is disabled. The default is false. A newly disabled link is redisplayed, without an anchor tag, to ensure that it is truly disabled from the user's point of view. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
href	URI of the content to display when the user clicks the *caption* text. If you want to invoke a client-side JavaScript method in the *href*, start the URI with `javascript:` as in: `href="javascript:zenPage.myMethod();"` The *href* value can be a literal string, or it can contain a Zen #()# runtime expression.
onclick	The *onclick* event handler for the link. Zen invokes this handler when the user clicks on the link. See "Zen Component Event Handlers." If *onclick* is specified, *href* is ignored. If *onclick* is not specified, the link displays the content specified by *href*.
style	CSS style to apply to cells in this link, for example: `color: red;`

Attribute	Description
target	String that controls where the new document is displayed when the user clicks on a link. In HTML, this is typically the name of a frame; however HTML also defines the following special values, which you can assign to the *target* attribute to get the desired behavior: • `"_blank"` — Open the link in a new window • `"_parent"` — Open the link in the parent window • `"_self"` — Open the link in the current window • `"_top"` — Open the link in the topmost window
title	Help message to display when the user hovers the cursor over the link, without clicking. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types." The *title* value can be a literal string, or it can contain a Zen #()# runtime expression.

One way to use <link> is to place it within an <expando> component to present a tree-style menu of links.

7.1.2 <locatorBar>

A <locatorBar> looks like the following example, based on the class ZENDemo.ActiveGroupDemo in the SAMPLES namespace. The <locatorBar> is the horizontal bar along the top of the illustration.



The corresponding <locatorBar> definition follows:

XML

```
<locatorBar id="locator" OnGetQuickLinks="GetQuickLinks">
  <locatorLink caption="Home" title="Home page"
               href="ZENDemo.Home.cls"/>
  <locatorLink caption="Active Group Demo"
               title="Active Group Demo" />
</locatorBar>
```

This <locatorBar> definition contains two <locatorLink> entries, which display in sequential order beginning at the far left end of the bar:

[Home] > [Active Group Demo]

The *caption* for each <locatorLink> is enclosed in square brackets and is separated by a right angle bracket from the next <locatorLink> in the sequence, from left to right. The user can display the *title* as a tooltip.

At the far right end of the bar is the drop-down list of quick links that this <locatorBar> has established by identifying an *OnGetQuickLinks* callback. The illustration shows that the user is hovering the cursor over this list and is about to make a selection. Doing so would cause the page identified by that link to display.

<locatorBar> has the following attributes:

Attribute	Description
Zen component attributes	<locatorBar> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
OnDrawBar	Name of a server-side callback method in the Zen page class. This method provides HTML content for the locator bar using &html<> syntax or WRITE commands. Zen invokes this method whenever it draws the locator bar, automatically passing it a %String that contains the *seed* value from the <locatorBar>. The callback must return a %Status data type. The following is a valid method signature: `Method DrawBar(pSeed As %String) As %Status` To use the above method as the callback, the developer would set `OnDrawBar="DrawBar"` for the <locatorBar>.
OnGetQuickLinks	Name of a server-side callback method in the Zen page class. This method defines a set of quick links to appear as a **Go to** drop-down list at the top right of the locator bar. For details, see the discussion following this table.
seed	Allows you to pass some arbitrary value to the *OnDrawBar* callback. The *seed* value can be a literal string, or it can contain a Zen #()# runtime expression.

The *OnGetQuickLinks* attribute provides the name of a server-side callback method in the Zen page class. This method must fill the array that is provided to it. If the method constructs a valid array, Zen displays each entry in the array as a **Go to** drop-down list at the top right of the locator bar. Zen invokes this method whenever it draws the locator bar, automatically passing in its single output parameter, an array subscripted by link caption. The callback must return a %Status value.

The following sample ObjectScript statement provides one entry for the array:

`Set pLink("`*caption*`")="`*uri*`"`

Where:

- *caption* is the label that appears on the link
- *uri* is the URI string, which may include query parameters in addition to the base URI

The following example shows a valid method signature and use of parameters. It generates the quick links shown in the illustration at the beginning of this topic. To use the following method as the callback, the developer would set `OnGetQuickLinks="GetLinks"` for the <locatorBar>.

Class Member

```
ClassMethod GetQuickLinks(Output pLinks) As %Status
{
  Set pLinks("Home") = "ZENDemo.Home.cls"
  Set pLinks("Expense Calculator") = "ZENDemo.ExpenseCalculator.cls"
  Set pLinks("MVC Master Detail") = "ZENMVC.MVCMasterDetail.cls"
  Set pLinks("MVC Chart") = "ZENMVC.MVCChart.cls"
  Set pLinks("MVC Meters") = "ZENMVC.MVCMeters.cls"
  Set pLinks("MVC Form") = "ZENMVC.MVCForm.cls"
  Set pLinks("Test Suite") = "ZENTest.HomePage.cls"
  Set pLinks("Controls") = "ZENDemo.ControlTest.cls"
  Set pLinks("Methods") = "ZENDemo.MethodTest.cls"
  Quit $$$OK
}
```

If you want to define the same set of quick links to use on more than one page in your application, you can define a method in your Zen application class and then have the callback in each page class invoke this application method, as follows. Note the dot (.) in front of the parameter *pLinks* in the call to the application class method. This is because it is an output parameter:

Class Member

```
ClassMethod GetQuickLinks(Output pLinks) As %Status
{
  #; dispatch to our application class
  Quit %application.GetQuickLinks(.pLinks)
}
```

7.1.3 <locatorLink>

<locatorLink> can appear only inside the <locatorBar> container. Each <locatorLink> defines one link within the navigation chain expressed in the <locatorBar>. <locatorLink> is the XML projection of the auxiliary class %ZEN.Auxiliary.locatorLink. <locatorLink> has the following attributes:

Attribute	Description
id	This value can be used to select a CSS style definition for the <locatorLink>.
name	Component name. Typically, this is not used for <locatorLink>.
caption	Text for this link in the <locatorBar>. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
href	URI of the content to display when the user clicks the *caption* text. If you want to invoke a client-side JavaScript method in the *href*, start the URI with `javascript:` as in: `href="javascript:zenPage.myMethod();"`
title	Help message to display when the user hovers the cursor over the link, without clicking. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

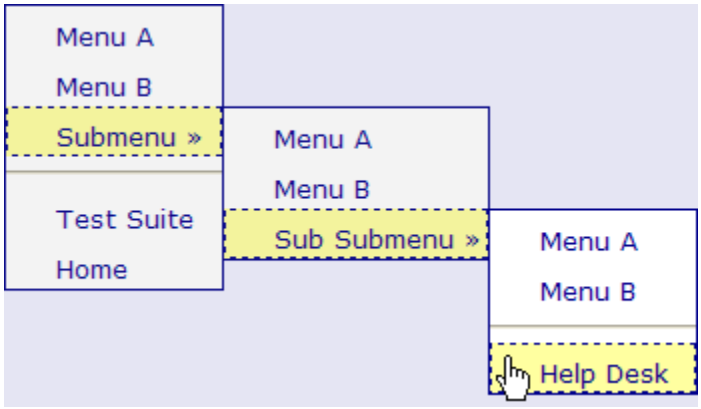
7.2 Menus

Menu components permit you to create classic navigation menus, with lists of choices from which the user can select an item by clicking on it. With each choice of a menu item, an event may occur depending on how the menu and menu item have been defined:

- A submenu might display
- A message might pop up
- A different page might display
- The contents of the current page might change

- Something else might happen, depending on how you have programmed the menu to respond when the user selects each <menuItem>

A vertical menu, expanded to three levels, looks like the following example, based on the class ZENTest.MenuTest in the SAMPLES namespace. Any options that produce submenus use the » right angle quote to indicate this, as shown for "Submenu" and "Sub Submenu".



The <menu> definition that produced this illustration is shown below:

XML

```
<menu id="menu2" layout="vertical">
  <menuItem caption="Menu A" link="javascript: alert('A');" />
  <menuItem caption="Menu B" link="javascript: alert('B');" />
  <menu id="menu2B" caption="Submenu" layout="vertical"
        onactivate="zenPage.activateMenu2B();">
    <menuItem caption="Menu A" link="javascript: alert('A');" id="menu2B_A" />
    <menuItem caption="Menu B" link="javascript: alert('B');" />
    <menu id="menu2BB" caption="Sub Submenu"  layout="vertical">
      <menuItem caption="Menu A" link="javascript: alert('A');" />
      <menuItem caption="Menu B" link="javascript: alert('B');" />
      <menuSeparator />
      <menuItem caption="Help Desk" link="ZENApp.HelpDesk.cls" />
    </menu>
  </menu>
  <menuSeparator />
  <menuItem caption="Test Suite" link="ZENTest.HomePage.cls" />
  <menuItem caption="Home" link="ZENDemo.Home.cls" />
</menu>
```

The next several sections explain details of the menu components in this example. For now, simply note:

- In many cases, the example uses JavaScript alerts in place of URI links for demonstration purposes. *link* values can take this form, or they can redirect the browser to other pages in the Zen application, as the example shows:
- A <menu> can contain a <menu>. If so, the contained <menu> becomes a submenu and its *caption* is listed in sequence along with any <menuItem> components at the same level. The above example provides three levels of <menu>.
- The example provides a <menuSeparator> on the first- and third-level menus. Note how the <menuSeparator> component generates the divider bar between "Submenu" and "Test Suite" and between "Sub Submenu" and "Help Desk" in the output.

Note: The "Client Side Menu Components" chapter in *Developing Zen Applications* describes more sophisticated menu components that are more easily fine-tuned and customized. Try the components in this chapter first, and if they do not suit your needs, consult the other book.

7.2.1 <menuItem>

Each item on a <menu> is defined using <menuItem>. <menuItem> has the following attributes. Since either of them may display their *caption* or *image* as a choice within a menu cell, <menuItem> and <menu> share all of these attributes in common. <menu> has additional attributes. For descriptions, see the <menu> section.

Table 7–1: Menu Cell Attributes

Attribute	Description
Zen component attributes	<menu> and <menuItem> have the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
caption	Text for this menu item when it appears as a choice within its containing <menu>. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
disabled	If true, this menu item is disabled. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
help	The *help* attribute supplies help text for this menu item. Generally what you want in place of *help* is the *title* attribute. *title* supplies tooltip text that displays whenever the user hovers the mouse over this item. All Zen components, including <menu> and <menuItem>, support the *title* attribute. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
image	Identifies the pathname and filename of an image to display for this menu item. If both *image* and *caption* are provided, they appear side by side, the image at left and the text at right. The *image* path is relative to the Caché installation directory. Typically this path identifies the images subdirectory for your Zen application, for example: `<image id="myFrame" src="/csp/myApp/images/myPic.png" />`
imageHeight	If an *image* is provided, *imageHeight* is its height in pixels. The default is 16.
imageWidth	If an *image* is provided, *imageWidth* is its width in pixels. The default is 16.
link	URI of the content to display when the user clicks the menu item. If you want to invoke a client-side JavaScript method in the link, start the URI with `javascript:` as in: `link="javascript:zenPage.myMethod();"`
linkResource	Name of a Caché resource. The user must have privileges to access this resource or this menu item becomes disabled. If you are not familiar with Caché resources, see the "Assets and Resources" chapter in the *Caché Security Administration Guide*.

Attribute	Description
onclick	The *onclick* event handler for the menu item. Zen invokes this handler when the user clicks on the menu item. See "Zen Component Event Handlers." If *onclick* is specified, *link* is ignored. If *onclick* is not specified, the menu item displays the content specified by *link*.
target	String that controls where the new document is displayed when the user clicks on a menu item. In HTML, this is typically the name of a frame; however HTML also defines the following special values, which you can assign to the *target* attribute to get the desired behavior: • `"_blank"` — Open the document in a new window • `"_parent"` — Open the document in the parent window • `"_self"` — Open the document in the current window • `"_top"` — Open document link in the topmost window

7.2.2 <menu>, <hmenu>, and <vmenu>

All Zen groups that do not have a predefined layout setting have a vertical layout by default. This is true for the <menu> component. However, it can be useful to explicitly state the desired layout of the menu group. You can do this by:

- Setting a value for the <menu> *layout* attribute (as in the example above)
- Choosing one of the following in place of <menu>:
  - <hmenu> — a horizontally oriented list of choices
  - <vmenu> — a vertically oriented list of choices

<menu>, <hmenu>, and <vmenu> have the following attributes:

Attribute	Description
Zen group attributes	A menu has the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*. However, the *layout* attribute does not apply to <hmenu> or <vmenu> because these components have *layout* set to "horizontal" and "vertical" respectively.
Zen menu cell attributes	A menu may display its *caption* or *image* as a choice within a menu cell. For this reason, <menu>, <hmenu>, and <vmenu> share several attributes in common with <menuItem>. For descriptions, see the <menuItem> section.
onactivate	The *onactivate* event handler for the menu. See "Zen Component Event Handlers." Used if this menu is a submenu. Zen invokes this handler just before the submenu is made visible.
onshowHelp	Client-side JavaScript expression that Zen invokes when the user moves the mouse over this menu item. Generally this expression invokes a client-side JavaScript method.

7.2.3 <menuSeparator>

A <menuSeparator> bar helps to visually group the options on a <menu>. <menuSeparator> has the same general-purpose attributes as any Zen component. For descriptions, see these sections:

- "Behavior" in the "Zen Component Concepts" chapter of *Using Zen*
- "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*

7.2.4 <accordionMenu>

An <accordionMenu> is a simple HTML5 accordion menu component. This component runs correctly only on HTML5 compliant browsers. <accordionMenu> supports the following attributes:

Attribute	Description
ongetdata	The ongetdata event handler. If it is defined, this event returns an array of items to be displayed in the menu.
onselect	The onselect event handler. If it is defined, this event is fired when the user clicks on an item in the menu.
selectedIndex	The currently selected item. This is a string of the form 'index1,index2,index3', where each index is the 0-based ordinal position of a menu, and its first and second level children. <accordionMenu> does not support more than three levels.
style	An additional style to apply to items in the menu.

You can get data to populate the <accordionMenu> either from an *ongetdata* event handler, or by using the *controllerId* property to connect the menu to an <altJSONProvider>. The following code samples show a simple <accordionMenu> that uses an <altJSONProvider> which gets data from an *OnGetArray* callback method.

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">
  <altJSONProvider id="jsonP" OnGetArray="GetData"/>
  <accordionMenu
    id="testMenu"
    selectedIndex="1,2"
    controllerId="jsonP"
    width="200"
    onselect="zen('sid').setProperty('value',key);"
  />
  <text id="sid" size="20"/>
</page>
```

Class Member

```
Method GetData(ByRef pParameters, Output pMetaData, Output pData) As %Status
{
  Set pMetaData = $LB("key","caption","image")

  Set pData(0) = $LB("1","Menu A","images/folderclosed.gif")
    Set pData(0,1) = $LB("0,1","Selection 1","images/file.png")
    Set pData(0,2) = $LB("0,2","Selection 2","images/file.png")
    Set pData(0,3) = $LB("0,3","Selection 3","images/file.png")
  Set pData(1) = $LB("1","Menu B","images/folderclosed.gif")
    Set pData(1,1) = $LB("1,1","Choice 1","images/file.png")
    Set pData(1,2) = $LB("1,2","Choice 2","images/file.png")
    Set pData(1,3) = $LB("1,3","Choice 3","images/file.png")
  Set pData(2) = $LB("1","Menu C","images/folderclosed.gif")
    Set pData(2,1) = $LB("2,1","Item 1","images/file.png")
    Set pData(2,2) = $LB("2,2","Item 2","images/file.png")
    Set pData(2,3) = $LB("2,3","Item 3","images/file.png")

  Quit $$$OK
}
```

The `OnGetArray` method uses the *pMetaData* argument to return a list of property names. You can use whatever property names you like, but also include in this list any or all of the following property names, which are given special handling by the <accordionMenu> component:

The OnGetArray Callback Method

- key – A value that identifies the selected menu item. Available to the *onselect* event handler.
- caption – Supplies text that appears in the menu item.
- action – A value that identifies the action to take when the user selects a menu item. Available to the *onselect* event handler.
- targetId – A value that identifies the object of the action. Available to the *onselect* event handler.
- image – Supplies the path to an image that appears in the menu item.
- children – An array of menu nodes that describe the submenu for a menu item.

<accordionMenu> passes the values provided for key, action, and targetId to the *onselect* event handler. The following simple example illustrates the availability of these values in the event handler.

Class Member

```
ClientMethod itemSelected(key, action, targetId) [ Language = javascript ]
{
  alert(key + " " + action + " " + targetId);
}
```

The text provided in the caption property appears as a label for the menu item. The image file supplied by the image property appears as part of the menu item label, to the left of the text. You do not generally need to specify the children property, as it is filled in by the <altJSONProvider> based on subscripts in the pData array provided by the `OnGetArray` callback.

7.3 Navigator

The <navigator> component creates a combination navigation and simple settings interface similar to that found on mobile devices. This is an HTML5 component. It works correctly only on HTML5 compliant browsers. The <navigator> component is implemented by the %ZEN.Component.navigator class.

The navigator component is completely driven by JavaScript data. The only Zen object needed to drive the navigator is the navigator component itself. The component can operate in the following modes:

- The top-level of the control is initially visible.
- The top-level is initially hidden and has to be made visible by a user event, such as pressing on an icon to make it appear.

<navigator> supports the following properties:

Attribute	Description
Zen group attributes	A menu has the same style and layout attributes as any Zen group. For descriptions, see “Group Layout and Style Attributes” in the “Zen Layout” chapter of *Using Zen*.
backgroundStyle	Style to apply to the navigator background.
columnWidth	Width of columns (in pixels). The default value is 320, which is also the value that gives the best visual results.
disclosureWidth	Width in pixels of the disclosure bar on the left. The attribute *showDisclosure* controls visibility of the disclosure bar. Clicking on the disclosure bar expands and contracts the navigator.

Attribute	Description
expanded	If true, then show the navigator. If false, the navigator is contracted, and not visible. The default value is true.
footerHeight	Height of the footer (in pixels). Set to 0 for no footer. The default value is 0. A value of 40 shows the footer with the best visual results.
headerHeight	Height of the header (in pixels). The default value is 40, which is also the value that gives the best visual results.
onarrange	The *onarrange* event handler. See "Zen Component Event Handlers." Zen invokes this handler when the order of items in the current sheet has changed. This event is passed 3 arguments: key, swap, and final. final is true when a value is finished changing. swap is an object with the property index and newIndex, containing the index of the item to move and its new location.
onbuttonclick	The *onbuttonclick* event handler. Zen invokes this handler when the user has clicks on a "header" or "footer" button.
onchange	The *onchange* event handler. Zen invokes this handler when a control within the property sheet has changed value. This event is passed 3 arguments: *key*, *value*, and *final*. *final* is true when a value is finished changing (such as when the user stops pressing a stepper button).
onclosebuttonclick	The *onclosebuttonclick* event handler. Zen invokes this handler when the user clicks on a "close" button for an item.
onexpand	The *onexpand* event handler. Zen invokes this handler when the user expands or contracts this component.
ongetcontent	The *ongetcontent* event handler: This defines the client-side code that defines the content of a "sheet" within this component. This is passed level, key, and value as arguments. This code should return an object with any of the following properties: • title – the title to display for the sheet. • url – if defined, the url to display as an iframe in the sheet (in the same domain). • html – custom html to display within the sheet. • childIndex – index number (0-based) of child of this component to display. • items – array of JavaScript objects used to define the contents.
onindent	The *onindent* event handler. Zen invokes this handler when the indentation level of an item in the current sheet has changed. This event is passed 3 arguments: key, list, and final. final is true when a value is finished changing. list is a an array containing the new ordinal positions of the items.
onpopupaction	The *onpopupaction* event handler. Zen invokes this handler when the user has invoked and applied a popup for an item.

Attribute	Description
onselect	The *onselect* event handler. Zen invokes this handler when a new choice has been selected within the property sheet. This handler is also called when a "drill" item is selected. This event is passed 3 arguments: *key*, *value*, and *which*. The value of *which* is "select" or "drill".
showDisclosure	If true, show the disclosure bar on the left. The attribute *disclosureWidth* controls width of the disclosure bar.

7.3.1 Creating and Sizing a <navigator>

Adding a navigator component to a page displays an empty navigator panel:

XML

```
<navigator id="navigator"/>
```

You can use CSS to control the size of the navigator, either in a style sheet or in JavaScript. For example, if the navigator has the id "navigator", the following code sets the height of the navigator to 600 pixels:

```
#navigator {
    height: 600px;
}
```

You can use the *headerHeight* attribute to control the height of the header banner:

XML

```
<navigator headerHeight="10"/>
```

You can also programmatically set the header height using the navigator's **setHeight** method, which also readjusts any internal geometry.

You can use the *columnWidth* attribute to control the width of the navigator. However, be aware that the navigator is designed for a fixed width of 320 pixels. The layout of menu controls does not display well at other widths.

You can use the *expanded* attribute to hide the navigator by completely collapsing it into a thin strip. The navigator defines an *onexpand* event that fires whenever the navigator is expanded or contracted. You can use this event to adjust the layout of other controls on the page when the navigator changes size.

7.3.2 Adding Content to the Navigator

The key to adding content is the *ongetcontent* event handler, which is called whenever the navigator needs to get content to display.

It is important to understand the stack-based nature of the navigator. You can think of the navigator as being a stack of playing cards. The navigator initially displays one card, and the *ongetcontent* event handler is called to get content for that card. If the user drills down into an item in the navigator, then a new card is placed on top of the previous card and *ongetcontent* is called again to get the contents of the new card. The user may then go back to the previous card or drill down yet another level.

To define content for the initial card (referred to as a sheet within the navigator API), define an *ongetcontent* handler:

XML

```
<navigator ongetcontent="return zenPage.getContentForLevel(level,key);"/>
```

Note that you need to use return so that the return value of the method is passed along. The callback method is passed two parameters:

- *level* – the 0-based stack level of the navigator
- *key* – the programmer-defined identifier for the current sheet. This is always `''` for the initial sheet.

Here is a basic implementation of an *ongetcontent* handler method:

Class Member

```
ClientMethod getContentForLevel(level, key) [ Language = javascript ]
{
    var content = { title: 'My Navigator', items:[] };
    return content;
}
```

The *ongetcontent* handler is expected to return an object with certain properties in it:

- *title* is the title that is displayed within the banner of the navigator.
- *items* is an array of items to display on the sheet.

The following example returns some content:

Class Member

```
ClientMethod getContentForLevel(level, key) [ Language = javascript ]
{
    var content = { title: 'My Navigator', items:[] };

    switch (key) {
    case '':
     // root
      content.items[content.items.length] =
       {display:'caption', caption:'Red'};
      content.items[content.items.length] =
       {display:'caption', caption:'Green'};
      content.items[content.items.length] =
       {display:'caption', caption:'Blue'};
     break;
    }

    return content;
}
```

key is `''` for the initial sheet. This example returns 3 items in the *items* array. Each item is an object with certain properties in it, in this case *display*, which specifies what to display and *caption*, the caption to display for each item.

At this point, the navigator does not do anything very interesting. You can use an *onselect* event handler to implement additional behavior. This event handler is called when the user selects an item:

```
<navigator ... onselect="alert(key);" ... />
```

This also has no effect, because there are no actions defined for the items in the navigator. Change the definition of the items so that they define *action* and *key*:

```
{display:'caption', caption:'Red', action:'select', key:'red'};
{display:'caption', caption:'Green', action:'select', key:'green'};
{display:'caption', caption:'Blue', action:'select', key:'blue'};
```

When the user clicks on an item in the navigator, the *onselect* action fires and passes along the key value.

The available values for *action* are:

- 'select' – fire the *onselect* callback.
- 'drill' – drill down one level.

- 'link' – navigate to a new URL, *value* supplies the URL.
- 'apply' – apply the value of the current item, which causes the navigator's *onchange* handler to fire.

The following additional item uses the 'drill' action:

```
content.items[content.items.length] =
 {display:'caption', caption:'Gray', action:'drill', key:'gray'};
```

The navigator now displays Gray as an option, with a drill-down icon. If we click on Gray, a new sheet appears on the navigator.

To supply content for this sheet, we need to modify our *ongetcontent* handler to add a case for when the value of *key* is 'gray':

```
switch(key) {
 case '':
   // root
   content.items[content.items.length] =
    {display:'caption', caption:'Red', action:'select', key:'red'};
   content.items[content.items.length] =
    {display:'caption', caption:'Green', action:'select', key:'green'};
   content.items[content.items.length] =
    {display:'caption', caption:'Blue', action:'select', key:'blue'};
   content.items[content.items.length] =
    {display:'caption', caption:'Gray', action:'drill', key:'gray'};
   break;
 case 'gray':
   // gray
   content.items[content.items.length] =
    {display:'caption', caption:'Black', action:'select', key:'black'};
   content.items[content.items.length] =
    {display:'caption', caption:'White', action:'select', key:'white'};
   break;
}
```

A more sophisticated *onselect* callback makes the page behavior more interesting:

```
<navigator ... onselect="zenPage.selectHandler(key);" ... />
```

Class Member

```
ClientMethod selectHandler(key) [ Language = javascript ]
{
 zenPage.getEnclosingDiv().style.background = key;
 while (zen('navigator').popSheet());
}
```

Now when the user selects an item, the color of the page changes, assuming that the value of *key* is a valid color value. The call to `popSheet()`, restores the navigator to its first sheet (keep popping until the stack level is back to 0).

You can also define a *value* for each item that is also passed to the *onselect* callback. In this case it is better to use *value* for the color name. In the following item definition, *value* specifies a darker shade of red:

```
{display:'caption', caption:'Red', action:'select', key:'red', value:'#330000'};
```

Pass *value* to the *onselect* callback, and use it to set the page color:

```
<navigator ... onselect="zenPage.selectHandler(key, value);" ... />
```

Class Member

```
ClientMethod selectHandler(key, value) [ Language = javascript ]
{
  zenPage.getEnclosingDiv().style.background = value;
  while (zen('navigator').popSheet());
}
```

7.3.3 Changing the Display and Appearance of Items

Each item can define a *display* attribute. This can take one of the following values:

- 'caption' – display the item caption
- 'caption-value-vt' – display the caption and value laid out vertically.
- 'caption-value-hz' – display the caption and value laid out horizontally.
- 'value' – display the item value Note that if the caption or value are too long, they are truncated.
- 'image-caption' – display an image and the caption
- 'image-caption-value-vt' – display an image and the caption and value laid out vertically.
- 'image-caption-value-hz' – display an image and the caption and value laid out horizontally.

The image is specified by the *image* property of the item:

```
{display:'caption-image', caption:'Red',
  image:'deepsee/blueprint_plan_48.gif', action:'select',
  key:'green', value:'green'};
```

If image is not defined, a default image is used. If you do not want an image, but want the same indent as if there were an image, then use `image:'none'`.

You can also add properties that control the style of an item:

- style – CSS style applied to the overall item (typically background).
- captionStyle – CSS style applied to the item caption.
- valueStyle – CSS style applied to the item value.
- disabled – if defined and true, then this item is disabled.
- selected – if defined and true, then display this item with the style defined for selected items.
- checked – if defined and true, then indicate that this item is checked

7.3.4 Editing Values in Items

A navigator can do more than act as a menu. You can use it to display and edit values using a small set of edit controls. These controls include:

- 'string' – display a simple text entry box
- 'slider' – display a slider control. *minValue* and *maxValue* can be used to define the range.
- 'slider-toggle' – display a slider control with a checkbox. If the checkbox is turned off the value is set to `""`.
- 'stepper' – display a up/down stepper control. *minValue* and *maxValue* can be used to define the range.
- 'stepper-value' – display a stepper with a value.
- 'switch' – display an on/off switch.
- 'choice' – display a small set of choices as buttons. *valueList* and *displayList* define the set of values and labels. This only works well for a small set of small choices due to the geometry of the navigator.

Perform the following steps to display an edit control:

- Set the value of the *display* property to something that displays a value, 'caption-value-hz' is a good choice.

- Set the value of the *edit* property to one of the available control types.

Here is an example of content items defining the various controls:

```
case 'edit':
 // edit controls
 content.items[content.items.length] =
  {display:'caption-value-hz', caption:'String',
   edit:'string', key:'string', value:''};
 content.items[content.items.length] =
  {display:'caption-value-hz', caption:'Slider',
   edit:'slider', key:'slider', value:0, minValue:0, maxValue:100};
 content.items[content.items.length] =
  {display:'caption-value-hz', caption:'Stepper',
   edit:'stepper', key:'stepper', value:0, minValue:0, maxValue:100};
 content.items[content.items.length] =
  {display:'caption-value-hz', caption:'Stepper 2',
   edit:'stepper-value', key:'stepper-value', value:0,
   minValue:0, maxValue:100};
 content.items[content.items.length] =
  {display:'caption-value-hz', caption:'Switch',
   edit:'switch', key:'switch', value:0};
 content.items[content.items.length] =
  {display:'caption-value-hz', caption:'Choice',
   edit:'choice', key:'choice', value:'box',
   valueList:'box,circle', displayList:'Box,Circle'};
 break;
```

Note that the *action* property is ignored if you define an *edit* property. *value* is the initial value of the control. It is not updated. You need to define an *onchange* callback to apply any changes and then make sure these values are used when the *ongetcontent* handler is next called.

The *onchange* callback is connected to the navigator component:

```
<navigator onchange="zenPage.dataChange(key,value,final);" ...
```

The *onchange* callback is passed 3 arguments: the *key* value for the item that has changed, the new *value*, and a *final* flag. The final flag is true if the value is done changing, and is relevant to sliders and steppers. It lets you track changes as they occur, or wait until they are complete.

7.3.5 Creating a Multiple Choice Item

If you want to let the user select a value from a set of choices and the *choice* control is too restrictive, you can create a sheet containing multiple choices and then let the user drill down to select one. The following example defines a simple list to select a political party. First, define a property on the page to hold the current choice:

```
Property party As %String;
```

The following item displays this property:

```
content.items[content.items.length] =
 {display:'caption-value-hz', caption:'Party', action:'drill',
  key:'party', value:this.party};
```

This item displays both the caption and the value, which is `this.party`. The action is *drill*. When the user drills down on this item, the *ongetcontent* handler asks for the contents for the sheet identified by the key 'party'. This sheet provides the list of political parties:

```
case 'party':
 content.items[content.items.length] =
  {display:'value', selected:this.party=='Democrat',
   value:'Democrat', action:'apply'};
 content.items[content.items.length] =
  {display:'value', selected:this.party=='Independent',
   value:'Independent', action:'apply'};
 content.items[content.items.length] =
  {display:'value', selected:this.party=='Libertarian',
   value:'Libertarian', action:'apply'};
 content.items[content.items.length] =
  {display:'value', selected:this.party=='Republican',
   value:'Republican', action:'apply'};
 break;
```

For each of these items, the action is 'apply'. When the user selects an option the *onchange* handler for the navigator is called and the sheet is popped off the stack. Note the use of *selected* to indicate the current value. The *onchange* handler updates the property `party`, so that when the original sheet is displayed, it shows the user's choice.

Class Member

```
ClientMethod dataChange(key, value, final) [ Language = javascript ]
{
 // apply change to data model
 switch (key) {
 case 'party':
  this.party = value;
  break;
 }
}
```

7.3.6 Displaying HTML

The content object returned by the *ongetcontent* handler can return arbitrary HTML to display in a navigator sheet. To do this, define a property called html in the content object, and set it to include valid HTML:

```
switch(key)
 case 'custom':
  content.html =
   '<font size="3" color="red">This is some text!</font>'
  content.title = 'HTML'
  break;
```

The html content fires data change events by invoking the navigator's **applyValue** method. This fires the *onchange* handler using the key value for the entire panel, which in this case is 'custom'.

7.4 Toolbar

The <toolbar> component is a versatile component that can display a series of different items along a horizontal bar. This is an HTML5 component. It works correctly only on HTML5 compliant browsers. The <toolbar> component is implemented by the %ZEN.Component.toolbar class.

<toolbar> has the following attributes:

Attribute	Description
imageStyle	Additional style to apply to images in the menu. Use this to change the size of images.

Attribute	Description
onchange	onchange event handler. Provides notification that a control in the toolbar has changed value. This event is passed the following arguments: key, value, and final. final is true when a value is finished changing.
ongetdata	ongetdata event handler. If defined, this event returns an array of items to be displayed in the menu.
onpagechange	onpagechange event handler. If defined, this event is fired when the user selects a new page number from a "pages" item. This event is passed the following arguments: key and page (selected page, 1-based). from the data element associated with the menu choice.
onselect	onselect event handler. If defined, this event is fired when the user clicks on a item in the menu. This event is passed the following arguments: key, action, and targetId from the data element associated with the menu choice.
scrollOffset	The index (0-based) of the first top-level item to be display when scrolled.
selectedIndex	The index (0-based) of the selected item in the top-level menu.
style	Additional style to apply to items in the menu.

You can display the following types of item in the toolbar:

- "item" – show a menu item. This is a caption that the user can click. An "item" may also define a list of children that are displayed as a drop-down menu. Clicking on an item raises the *onselect* event.
- "tab" – show a "tab". This is a caption that the user can click. When the user clicks a tab, it becomes the current tab and all other tab items are unselected. Clicking on a tab raises the *onselect* event.
- "message" – display a text message.
- "choice" – display a set of choices (using a displayList and valueList) as a choice button.
- "pages" – display a set of paging buttons, such as may be present on a search results page. In this case you can specify the minValue (first page #), maxValue (last page #) and value (current page). Clicking on a page invokes the onchangepage event.

You define the contents of the toolbar with a JavaScript object, which you can supply with the *ongetdata* event handler or by connecting to an <altJSONProvider>. The JavaScript object is assumed to have a collection named children that define a set of objects that specify the items in the toolbar. Each of these objects has a *type* property that indicates what type of item to display. The *caption* is the caption. and *key* is a key used to identify an item, such as in a callback event.

For example, the following Zen page and *ongetdata* event handler create an example toolbar:

Class Member

```
XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
  <page xmlns="http://www.intersystems.com/zen" title="">
    <toolbar label="Test Toolbar" width="900px"
     ongetdata="return zenPage.myData();" />
  </page>
}
```

Class Member

```
ClientMethod myData() [ Language = javascript ]
{
  var data = {
  children:[
  { caption:"Home", key:"menuHome", type:'item' },
  { caption:"Menu", key:"menuMenu", type:'item',
```

```
  children:[
  { caption:"Menu 1", key:"menu1", type:'item'},
  { caption:"Menu 2", key:"menu2", type:'item'},
  { caption:"Menu 3", key:"menu3", type:'item'}
  ]
  },
  { caption:"", key:"menuChoice", type:'choice', valueList:'On,Maybe,Off' },

  { caption:"Tab 1", key:"menuTab1", type:'tab' },
  { caption:"Tab 2", key:"menuTab2", type:'tab', selected:true },
  { caption:"Tab 3", key:"menuTab3", type:'tab' },
  { caption:"This is a message", key:"menuMsg", type:'message' },
  { html:'<input type="text"/>', key:"menuSearch", type:'item' },
  { caption:'Page', key:"menuPages", type:'pages', minValue:1,maxValue:20,value:15 },
    ]
  };
  return data;
}
```

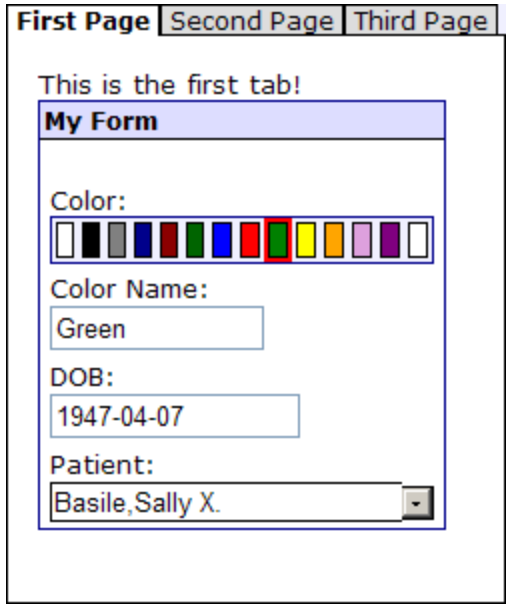
7.5 Tabs

The following Zen components support tabbed menus and forms for Zen applications:

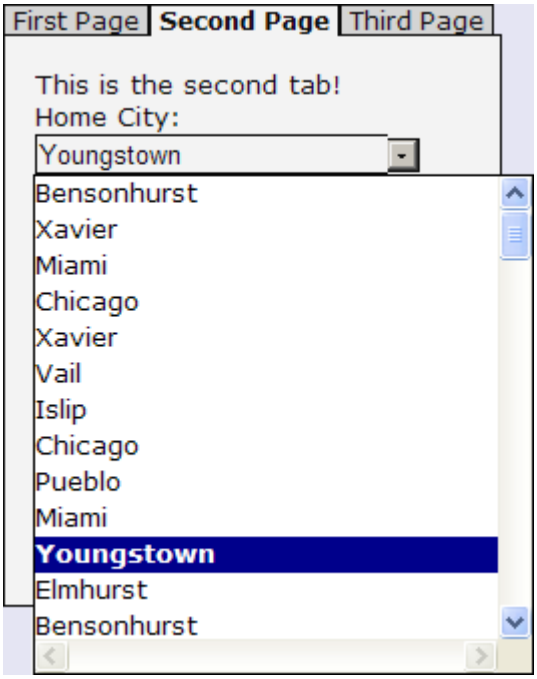
- "<tabGroup>"
- "<lookoutMenu>"
- "<tab>"

7.5.1 <tabGroup>

A <tabGroup> displays a set of <tab> components. It can only display one <tab> at a time. The user can choose one of these tabs to be displayed on top of the set. A <tabGroup> looks like the following example, based on the class ZENTest.TabTest in the SAMPLES namespace.



In the above example, the user has clicked the tab labelled "First Page." In the following example, the user has clicked the tab labelled "Second Page" and has clicked the down-arrow to select a "Home City" option from the <dataCombo> control.



The <tabGroup> definition that produces these illustrations is shown below:

XML

```
<tabGroup id="tabGroup" showTabBar="true"
          onshowTab="zenPage.updateButtons();" remember="true">
  <tab caption="First Page">
    <hgroup>
      <spacer width="15" />
      <vgroup>
        <spacer height="5"/>
        <html>This is the first tab!</html>
        <form width="75%" layout="vertical"
              cellStyle="padding: 2px; padding-left: 5px; padding-right: 5px;"
              groupStyle="border:1px solid darkblue;">
          <titleBox title="My Form" titleStyle="background: #DDDDFF;"
                    containerStyle="padding: 0px;" />
          <spacer height="5"/>
          <colorPicker title="This is a custom control!"
                       label="Color:" name="Color" />
          <text label="Color Name:" name="ColorName" size="12" />
          <text label="DOB:" id="DOB" name="DOB" size="15"
                maxlength="10" valign="bottom"/>
          <dataCombo label="Patient:" name="Patient" size="24"
                     sql="SELECT Name FROM ZENDemo_Data.Employee
                          WHERE Name %STARTSWITH ? ORDER BY Name"/>
        </form>
      </vgroup>
    </hgroup>
  </tab>
  <tab caption="Second Page">
    <spacer height="5"/>
    <html>This is the second tab!</html>
    <dataCombo label="Home City:" name="City" size="24"
               sql="SELECT Location FROM ZENApp_Data.Customer
                    WHERE Location %STARTSWITH ? ORDER BY Name"/>
  </tab>
  <tab caption="Third Page" tabResource="MyResource">
    <spacer height="5"/>
    <html>This is the third tab!</html>
    <dynaGrid id="grid">
      <gridColumn label="Name"   width="25%" />
      <gridColumn label="Salary" width="25%"  />
      <gridColumn label="Comment" width="50%" />
      <gridRow label="R1" />
      <gridRow label="R2" />
      <gridRow label="R3" />
    </dynaGrid>
  </tab>
</tabGroup>
```

When a <tabGroup> has contents that do not fit within its defined height or width, Zen clips the display of excess content at the right and bottom edges of the defined <tabGroup> size, leaving room for a horizontal or vertical scrollbar that allows the user to scroll to view the contents. Application and page level stylesheets can override this default behavior by applying the CSS property `overflow` to the element class `tabGroupBody`.

<tabGroup> has the following attributes. Since either <tabGroup> or <lookoutMenu> may display <tab> components, both <tabGroup> and <lookoutMenu> share these attributes. <lookoutMenu> has additional attributes. For descriptions, see the <lookoutMenu> section.

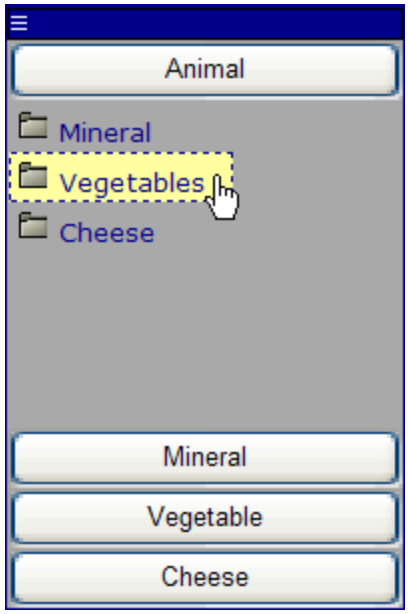
Table 7–2: Tab Group Attributes

Attribute	Description
Zen group attributes	<tabGroup> and <lookoutMenu> each have the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*. For <tabGroup>, the *height* attribute has no effect; it is necessary to control the height of the <tabGroup> from the CSS stylesheet.
currTab	1–based sequential number of the tab currently displayed. The default is 1. The *currTab* value may contain Zen #()# runtime expressions.
onhideTab	The *onhideTab* event handler for the menu item. Zen invokes this handler when a previously displayed tab becomes hidden. See "Zen Component Event Handlers."
onshowTab	Client-side JavaScript expression that Zen invokes when a previously hidden tab becomes the displayed tab. Generally this expression invokes a client-side JavaScript method.
remember	If true, remember the most recently displayed tab number in a session cookie and return to this tab when redisplayed. The default is false. *remember* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showBody	If true, display the set of tabs that belong to this <tabGroup>. By setting *showBody* to false, and setting *showTabBar* true, you can display a set of tab bar buttons with no tab contents underneath. The default for *showBody* is true, and for *showTabBar* is false. *showBody* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
showTabBar	If true, display a set of tab buttons along the top of this group. The default is false. *showTabBar* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

7.5.2 <lookoutMenu>

A <lookoutMenu> presents a stack of buttons, one for each tab. Clicking on a button shifts the other buttons down so that they display below the contents of the currently selected tab.

The following example is based on the class ZENTest.LookoutMenuTest in the SAMPLES namespace. In the illustration, the user has clicked the "Animal" button to reveal the contents of that tab. This selection moved the remaining buttons to the bottom of the lookout menu. The cursor is now on the "Vegetables" menu item. If the user clicks this item, the content of its *link* is activated as defined in the corresponding <menuItem>.



A <lookoutMenu> contains a set of <tab> components, which in turn contain <menuItem> and <menuSeparator> components. These tabs are groups that can contain any component, but when they are present inside a <lookoutMenu> they contain the components that would normally define a <menu>.

The <lookoutMenu> definition that produced the above illustration follows:

XML

```
<lookoutMenu id="lookout" expandable="true">
  <tab caption="Animal" id="animal">
    <menuItem caption="Mineral"
              link="javascript: zenPage.toggleTab('mineral');"
              image="images/folder.gif" />
    <menuItem caption="Vegetables"
              link="javascript: zenPage.toggleTab('vegetable');"
              image="images/folder.gif" />
    <menuItem caption="Cheese"
              link="javascript: zenPage.toggleTab('cheese');"
              image="images/folder.gif" />
  </tab>
  <tab caption="Mineral" id="mineral" tabResource="MyResource">
    <form>
      <text label="Name:" />
      <text label="Weight:" />
    </form>
  </tab>
  <tab caption="Vegetable" id="vegetable" disabled="false">
    <menuItem caption="Menu A"
              link="javascript: alert('A');"
              help="Option A" image="images/folder.gif" />
    <menuItem caption="Menu B"
              link="javascript: alert('B');"
              help="Option B" image="images/folder.gif" />
    <menuSeparator />
    <menuItem caption="Disable"
              link="javascript: zenPage.toggleTab('vegetable');"
              image="images/folder.gif" />
  </tab>
  <tab caption="Cheese" id="cheese">
    <menuItem caption="Menu C" link="javascript: alert('C');"
              help="Option C" image="images/folder.gif" />
  </tab>
</lookoutMenu>
```

<lookoutMenu> has the following attributes:

Attribute	Description
Zen tab group attributes	<lookoutMenu> has the same general-purpose attributes as <tabGroup>. For descriptions, see the "Tab Group Attributes" table in the <tabGroup> section.
expandable	If true, this <lookoutMenu> group is expandable. The default is false. When *expandable* is true, an expansion bar displays along the top of the <lookoutMenu> as shown in the illustration above. The user clicks on this bar to toggle the *expanded* state of the <lookoutMenu>. When the menu is contracted, only the expansion bar is visible. The user clicks the bar again to expand the <lookoutMenu>. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
expanded	If true, this <lookoutMenu> group is expanded (children visible). If false, it is contracted (children not visible). The default is true. *expanded* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
oncontract	The *oncontract* event handler for the <lookoutMenu>. Zen invokes this handler just before contracting (hiding) the children of this <lookoutMenu> group. See "Zen Component Event Handlers."
onexpand	Client-side JavaScript expression that Zen invokes just before expanding (displaying) the children of this <lookoutMenu> group.

7.5.3 <tab>

A <tab> is a specialized group that defines a tab within a <tabGroup> or <lookoutMenu>. <tab> has the following attributes:

Attribute	Description
Zen group attributes	<tab> has the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*. A <tabGroup> or <lookoutMenu> uses the *hidden* attribute to make the current tab visible and all other tabs invisible. That is, when a <tabGroup> or <lookoutMenu> is displayed, it sets the *hidden* attribute for the currently visible <tab> to false and sets all the others to true. For this reason, explicitly setting *hidden* for a <tab> element has no effect. If you need to prevent users from accessing a <tab> you can set its *disabled* attribute to true.
caption	Text for this tab when it appears as a choice within its containing <tabGroup> or <lookoutMenu>. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
link	An optional URI value. If defined, the tab displayed within the <tabGroup> bar becomes a link referring to this URI. No link is displayed for the current tab. The *link* value may contain Zen #()# runtime expressions.
tabResource	Name of a Caché resource. The user must have privileges to access this resource or this tab becomes disabled. If you are not familiar with Caché resources, see the "Assets and Resources" chapter in the *Caché Security Administration Guide*.

7.6 Trees

The following Zen components each provide a hierarchical outline of links. The outline expands or contracts in response to user clicks:

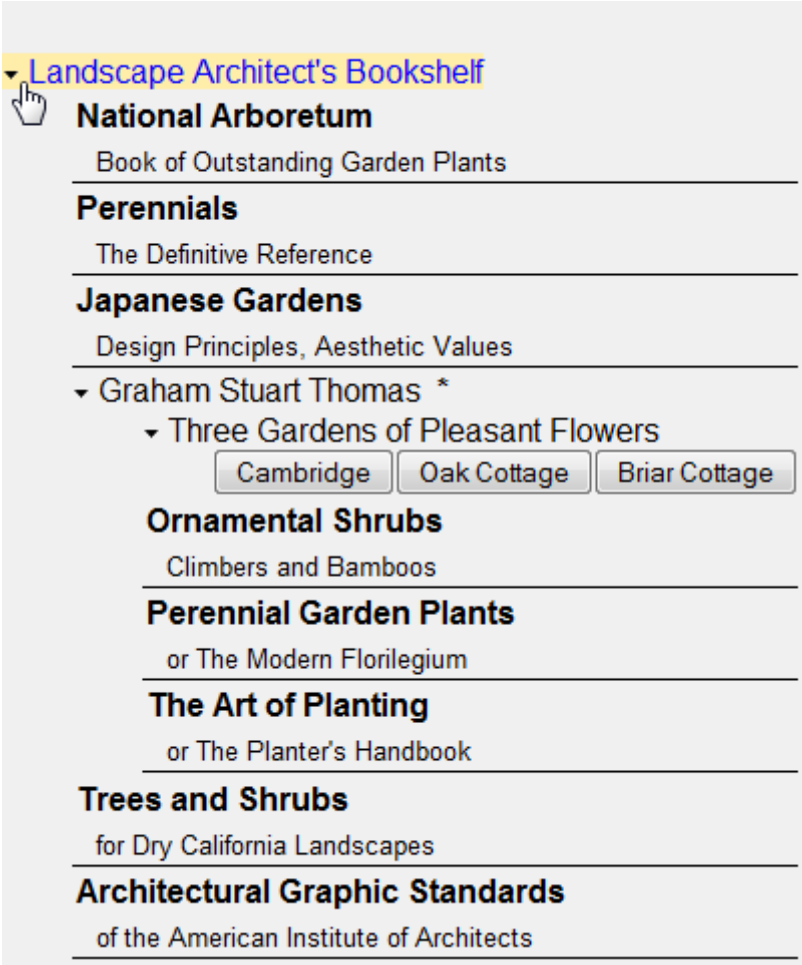
- <expando>
- <dynaTree>

7.6.1 <expando>

The <expando> is a group with the ability to show or hide its children. A right-arrow graphic (‣) beside the <expando> label indicates that the group is expanded (displayed), and a down-arrow graphic (▾) indicates that the group is contracted (hidden). A fully contracted <expando> group looks like the following example.

The user expands the group by clicking on its label. The contents of the <expando> then display below the label. As a Zen group, an <expando> may contain any type of Zen component. In particular, each level of the <expando> may contain additional, nested <expando> groups that can be expanded or contracted independently of the containing <expando> group.

In the following example, the user has just clicked on the label to expand the group. The cursor is now poised to contract the <expando> by clicking the label again. This example contains three nested <expando> groups. The innermost <expando> uses a horizontal layout, and contains <button> components instead of <titleBox> components as in the other levels.



The <expando> definition that produced this example follows:

XML

```
<expando caption="Landscape Architect's Bookshelf"
         childIndent="35px" remember="true">
   <spacer height="1" />
   <titleBox title="National Arboretum"
             subtitle="Book of Outstanding Garden Plants" />
   <titleBox title="Perennials"
             subtitle="The Definitive Reference" />
   <titleBox title="Japanese Gardens"
             subtitle="Design Principles, Aesthetic Values" />
   <spacer height="1" />

   <expando caption="Graham Stuart Thomas" OnDrawContent="DrawContent"
            childIndent="35px" remember="true">

      <expando caption="Three Gardens of Pleasant Flowers"
               layout="horizontal" childIndent="35px" remember="true">
         <button caption="Cambridge" onclick="zenPage.cambridgeClick()"/>
         <button caption="Oak Cottage" onclick="zenPage.oakCottageClick()"/>
         <button caption="Briar Cottage" onclick="zenPage.briarCottageClick()"/>
      </expando>

      <titleBox title="Ornamental Shrubs"
                subtitle="Climbers and Bamboos" />
      <titleBox title="Perennial Garden Plants"
                subtitle="or The Modern Florilegium" />
      <titleBox title="The Art of Planting"
```

```
                subtitle="or The Planter's Handbook" />
   </expando>

   <spacer height="1" />
   <titleBox title="Trees and Shrubs"
             subtitle="for Dry California Landscapes" />
   <titleBox title="Architectural Graphic Standards"
             subtitle="of the American Institute of Architects" />
</expando>
```

This definition references a number of methods defined in the page class. For example, the second-level <expando> references the server-side callback method **DrawContent** to produce additional formatted text, in this case a list of honors awarded to the author of the books listed. **DrawContent** looks like this:

Class Member

```
Method DrawContent(ByRef expando As %ZEN.Component.expando) As %Status
{
  &html<<b> (OBE, VMH, DHM, VMM)</b>>
  Quit $$$OK
}
```

The <expando> component has the following attributes:

Attribute	Description
Zen group attributes	<expando> has the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*. The default *layout* for <expando> is vertical.
animate	If true, Zen animates the appearance and disappearance of the <expando> group contents. If false, it does not. *animate* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
caption	Text to display as the title of this <expando> group. This text displays even when the <expando> is contracted. This text is *not* automatically HTML escaped. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
childIndent	HTML length value giving the amount to indent the children in the expanded list of items. The above example uses `35px`. The default is to use no indentation, aligning the children with the *caption*.
expanded	If true, this <expando> group is expanded (children visible). If false, it is contracted (children not visible). The default is true. *expanded* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

Attribute	Description
framed	If true, display a border around the entire group and display the *caption* within a more formal title box. This more formal version of the <expando> component is also known as a *disclosure*. The default is false. *framed* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types." The %ZEN.Component.expando class offers a server-side callback method **%OnDrawTitleOptions** which, if defined in a subclass of %ZEN.Component.expando, provides a way to add content to the right side of the title bar when *framed* is true. Any HTML written by **%OnDrawTitleOptions** is injected into the title bar when the <expando> is displayed.
imageContracted	URI of the image to display when the <expando> group is contracted. The path that you provide for *imageContracted* must be relative to the CSP directory in the Caché installation directory. The default for *imageContracted* is a right-arrow graphic (‣) located at the URI `broker/images/disclosure-contracted.gif`.
imageExpanded	URI of the image to display when the <expando> group is expanded. The path that you provide for *imageExpanded* must be relative to the CSP directory in the Caché installation directory. The default for *imageExpanded* is a down-arrow graphic (▾) located at the URI `broker/images/disclosure-expanded.gif`.
oncontract	The *oncontract* event handler for the <expando>. Zen invokes this handler just before contracting (hiding) the children of this <expando> group. See "Zen Component Event Handlers."
OnDrawContent	Name of a server-side callback method in the Zen page class. This method injects HTML content into the <expando> using &html<> syntax or WRITE commands. Zen invokes this method whenever it draws the <expando>, automatically passing it a %String that contains the *seed* value from the <expando>. The callback must return a %Status data type. The following is a valid method signature: `Method DrawMe(pSeed As %String) As %Status` To use the above method as the callback, the developer would set `OnDrawContent="DrawMe"` for the <expando>.
onexpand	Client-side JavaScript expression that Zen invokes just before expanding (displaying) the children of this <expando> group.
remember	If true, remember the most recent expanded state in a session cookie and return to this state when redisplayed. The default is false. *remember* has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."

<expando> can be particularly useful in navigation when combined with <link>.

7.6.2 <dynaTree>

The <dynaTree> component displays a hierarchical collection of user-defined items as an expandable tree. In many ways <dynaTree> is similar to <expando>. However, instead of specifying the components that it contains, <dynaTree> acquires its contents dynamically at runtime. You can provide data for a <dynaTree> as follows:

- Use the "dataGlobal attribute" to specify a global whose data supplies the contents of the tree.
- Use the "OnGetNodeInfo callback" to get data for each node within the tree.
- Use the "OnGetTreeInfo callback" to fill in a local array that supplies the contents of the tree.

Note that <dynaTree> builds the tree from the root of the global or array. It does not support starting the tree from a node other than the base node.

7.6.2.1 <dynaTree> Using a Global

The simplest way to define a <dynaTree> is to use a Caché global (multidimensional array). For background information about globals, refer to the book *Using Caché Globals*.

The following <dynaTree> definition references the ^myTree global by providing the *dataGlobal* attribute:

XML

```
<dynaTree id="tree" dataGlobal="^myTree" />
```

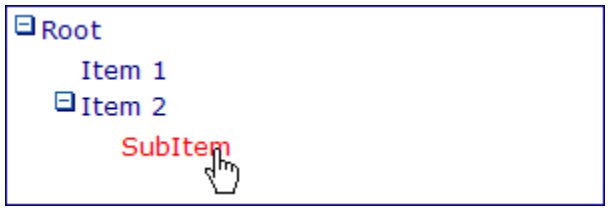
This definition causes the contents of the ^myTree global to be displayed on the page as nodes within an expandable tree.

Node Labels

Suppose you define ^myTree using the following statements at the Terminal command line:

```
Set ^myTree("Root","Item 1") = "ZENMVC.MVCForm.cls"
Set ^myTree("Root","Item 2") = "http://www.intersystems.com"
Set ^myTree("Root","Item 2","SubItem") = "ZENDemo.Home.cls"
```

Each global node now has one or more subscripts, such as `"Root"` or `"Item 2"`. Zen uses these subscripts as the display values for nodes in the <dynaTree>. The logical value for each node is the value of the global at that subscript. Our ^myTree global can now generate a <dynaTree> that looks like the following illustration. This <dynaTree> is fully expanded, with the user holding the cursor over the node `"SubItem"` at the lowest level. Based on the previous SET statements, the logical value for this node is `"ZENDemo.Home.cls"` and its display value is `"SubItem"`



When you use a global to define a <dynaTree>, the resulting nodes always appear in alphabetical order (by label) because that is the way globals are organized in the database.

Node Values

When the user clicks on a <dynaTree> node label such as:

`"SubItem"`

The corresponding node value is triggered. If the values are set as described in the "Node Labels" section of this chapter, then this value is:

```
"ZENDemo.Home.cls"
```

Each node value is a string. Typically the node values are URI values. For example:

- The name of a Zen page class:

  ```
  "ZENMVC.MVCForm.cls"
  ```

- A web site:

  ```
  "http://www.intersystems.com"
  ```

- Or the URI of some other content.

If you want a link to invoke JavaScript, you can start the URI with the string `javascript:` as in the following examples:

- You can invoke a client-side JavaScript method in the page class:

  ```
  "javascript: zenPage.myMethod();"
  ```

- Or simply execute a JavaScript expression:

  ```
  "javascript: alert('You clicked me!');"
  ```

  When providing a JavaScript expression, use double quotes to enclose the value and single quotes (if needed) inside the expression, as shown above.

7.6.2.2 Parameters for <dynaTree> Callback Methods

A <dynaTree> can contain zero or more <parameter> elements. Each <parameter> specifies an input parameter for the callback method that generates the <dynaTree>. This callback method might be:

- "OnGetNodeInfo" to get data for each node within the tree.
- "OnGetTreeInfo" to fill in a local array that supplies the contents of the tree.

The <parameter> element has the following attributes:

Attribute	Description
paramName	The *paramName* must be unique within the <dynaTree>. It becomes a subscript in the array of parameters passed to the callback method.
value	The *value* supplied for a <parameter> can be a literal string, or it can contain a Zen #()# runtime expression.

7.6.2.3 <dynaTree> OnGetNodeInfo Callback Method

<dynaTree> can get its list of nodes by invoking a server-side callback method defined in the page class. The method name is specified using the *OnGetNodeInfo* attribute. For example:

XML

```
<dynaTree id="myTree"
          OnGetNodeInfo="GetNodeInfo"
          onclick="zenPage.treeClick(zenThis);" >
  <parameter paramName="a" value="10" />
  <parameter paramName="b" value="20" />
</dynaTree>
```

This example defines a tree whose nodes are provided by the server-side callback method **GetNodeInfo**. When the user clicks on an item, the **treeClick** method is called. The example also provides two parameters for the **GetNodeInfo** method.

The *OnGetNodeInfo* callback is called repeatedly to get information about each node displayed within the tree. Zen handles this repetition as follows:

- A tree contains a set of top-level nodes at level 1. A node may contain child nodes; each child is considered to have a level number one greater than its parent: 2, 3, 4, etc.
- Starting with level 1, Zen calls the *OnGetNodeInfo* callback repeatedly until it returns false, indicating that there are no more nodes at the current level.
- If a node reports that it has child nodes, Zen calls the *OnGetNodeInfo* callback repeatedly (with the appropriate value for *pLevel*) to get information on each child node until false is returned for that child level.

The *OnGetNodeInfo* callback method must have a signature that looks like this:

```
Method GetNodeInfo(Output tSC As %Status,
    ByRef pParams As %String,
    pLevel As %Integer,
    ByRef pHandle As %String,
    pNodeInfo As %ZEN.Auxiliary.NodeInfo) As %Boolean
```

Where:

- The method returns a %Boolean value indicating whether or not there is a node at the level indicated by *pLevel*.
- *tSC* is a status code, returned by reference, that indicates success.
- *pParms* represents any <parameter> elements defined by the <dynaTree>. *pParms* is an array. Each member of this array uses its *paramName* as a subscript and its *value* as a value.
- *pLevel* is the current level of the tree.
- *pHandle* is user-definable value that is passed, unchanged, to each call to the callback. The callback can use this store any state information it needs while providing the view information.
- *pNodeInfo* is a pre-instantiated %ZEN.Auxiliary.NodeInfo object and is used to specify how the tree node should be displayed. A %ZEN.Auxiliary.NodeInfo object has the following properties.

Property	Description
expanded	If true, this node (if it has children) is initially displayed as expanded. If false, it is initially displayed as contracted. The default is true. expanded has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types." It has the value 1 or 0 in server-side code, true or false in client-side code.
hasChildren	If true, this node has one or more child nodes. If this is the case, the next call to the OnGetNodeInfo callback fetches information about these child nodes. The default is true. hasChildren has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types." It has the value 1 or 0 in server-side code, true or false in client-side code.

Property	Description
link	If non-empty, link defines a link that is used if the user clicks on this node. This can be a URI such as the name of a page to display, or a JavaScript expression. If you want to invoke a client-side JavaScript method in the link, start the URI with `javascript:` as in: `link="javascript:zenPage.myMethod();"` When providing a JavaScript expression, use double quotes to enclose the value and single quotes (if needed) inside the expression.
text	A string containing a value to display for this node within the tree.
value	A string containing a logical value to associate with this node within the tree.

The following code example provides a complete *OnGetNodeInfo* method.

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
<page xmlns="http://www.intersystems.com/zen" title="">
<dynaTree id="myTree" OnGetNodeInfo="GetNodeInfo">
</dynaTree>
</page>
}

Method GetNodeInfo(
    Output tSC As %Status, ByRef pParams As %String,
    pLevel As %Integer, ByRef pHandle As %String,
    pNodeInfo As %ZEN.Auxiliary.NodeInfo) As %Boolean
{
    // We store our personal state information in pHandle that is passed,
    // unchanged, to each call to the callback.
    // In the beginning: pHandle = ""
    If pHandle = ""
    {
        Set pHandle = 1
    }
    // Let's create 10 nodes
    ElseIf (pHandle = 10)
    {
        Quit 0
    }
    // Set the node count as name and value
    Set pNodeInfo.value = "Node "_pHandle
    Set pNodeInfo.text = "test"_pHandle
    Set pNodeInfo.expanded = 1

    // We add subnode within the 3rd node...
    If (pHandle = 3)
    {
        Set pNodeInfo.hasChildren = 1
    }
    // ...and also subnode for the 6th subnode of the 3rd node.
    ElseIf (pHandle = 6)
    {
        Set pNodeInfo.hasChildren = 1
    }

    Set pHandle = pHandle + 1

    Quit 1
  }
```

7.6.2.4 <dynaTree> OnGetTreeInfo Callback Method

<dynaTree> can get its complete tree as an multidimensional array of node information, by invoking a server-side callback method defined in the page class. The method name is specified using the *OnGetTreeInfo* attribute. For example:

XML

```
<dynaTree id="tree"
          OnGetTreeInfo="GetTreeInfo"
          onclick="zenPage.treeClick(zenThis);">
  <parameter paramName="count" value="20"/>
</dynaTree>
```

The *OnGetTreeInfo* callback method must have a signature that looks like this:

```
ClassMethod GetTreeInfo(pRoot As %String, Output pTree, ByRef pParms) As %Status
```

Where:

- The method returns a status code.
- *pParms* represents any <parameter> elements defined by the <dynaTree>. *pParms* is an array. Each member of this array uses its *paramName* as a subscript and its *value* as a value.
- *pRoot* is the display name of the node to be loaded.

The previous example defines a <dynaTree> whose data is provided by the server-side callback method **GetTreeInfo**. When the user clicks on an item, the **treeClick** method is called with the value 20 in its parameter array subscripted by the key `"count"`.

Important: If you use *OnGetTreeInfo* to define the <dynaTree> this excludes using the *OnGetNodeInfo* or *dataGlobal* attributes in the same <dynaTree>.

The following example implements **GetTreeInfo** from the previous example. This shows the required method signature for an *OnGetTreeInfo* callback. For further examples, see the class ZENTest.DynaTreeTest in the SAMPLES namespace. In that example there are several different *OnGetTreeInfo* callbacks. The user clicks a radio button to choose which of these callbacks is used to generate the <dynaTree> on this page.

Class Member

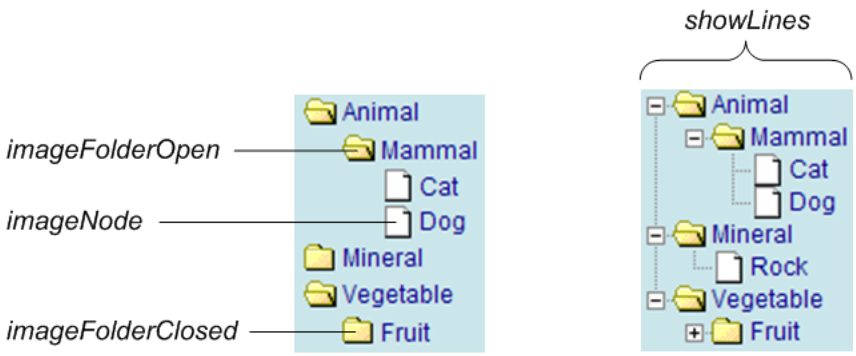
```
ClassMethod GetTreeInfo(pRoot As %String, Output pTree, ByRef pParms) As %Status
{
  if pRoot=""
  {
    #; top-most nodes are children of 0
    Set pTree(0,"ch",1) = ""
    Set pTree(0,"ch",2) = ""
    Set pTree(0,"ch",3) = ""

    #; each node supplies: $LB(caption, value, hasChildren, link, expanded, icon)
    Set pTree(1) = $LB("Animal","Animal",1,"",1,,"General types of animal")
    Set pTree(2) = $LB("Mineral","Mineral",1,"",1,,"General types of mineral")
    Set pTree(3) = $LB("Vegetable","Vegetable",1,"",1,,"General types of vegetable")
  }
  elseif pRoot="Animal" //id 1
  {
    Set pTree(4) = $LB("Mammal","Mammal",1,"",1)
    Set pTree(0,"ch",4) = ""
  }
  elseif pRoot="Mineral" //id 2
  {
    Set pTree(7) = $LB("Rock","Rock",0,"",1)
    Set pTree(0,"ch",7) = ""
  }
  elseif pRoot="Vegetable" //id 3
  {
    Set pTree(8) = $LB("Fruit","Fruit",1,"",1)
    Set pTree(0,"ch",8) = ""
  }
  elseif pRoot="Mammal" //id 4
  {
    Set pTree(5) = $LB("Cat","Cat",0,"",1)
    Set pTree(6) = $LB("Dog","Dog",0,"",1)
    Set pTree(0,"ch",5) = ""
    Set pTree(0,"ch",6) = ""
  }
  elseif pRoot="Fruit" //id 8
```

```
  {
    Set pTree(9) = $LB("Apple","Apple",0,"",1)
    Set pTree(10) = $LB("Banana","Banana",0,"",1)
    Set pTree(11) = $LB("Cherry","Cherry",0,"",1)
    Set pTree(0,"ch",9) = ""
    Set pTree(0,"ch",10) = ""
    Set pTree(0,"ch",11) = ""
  }
  Quit $$$OK
}
```

7.6.2.5 <dynaTree> Attributes

The <dynaTree> component is the XML projection of the %ZEN.Component.dynaTree class. This topic has already described the *dataGlobal*, *OnGetNodeInfo*, and *OnGetTreeInfo* attributes in detail. The following table lists all the <dynaTree> attributes.

The purpose of many <dynaTree> attributes is to configure the images that the user clicks to control expansion and contraction. The following diagram shows many of the images that the <dynaTree> displays while the component is in use. The following table describes how to configure these images, if you wish to substitute your own images instead.



Attribute	Description
Zen component attributes	<dynaTree> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
childIndent	HTML length value giving the amount by which each level within the dynaTree should be indented. The default is to use no indentation. *childIndent* applies only when *showLines* is false. When *showLines* is true, indentation is controlled by the <dynaTree> to fit the lines that it provides.
dataGlobal	Name of a Caché global (multidimensional array) that can provide the contents of the <dynaTree>. This string must include the ^ prefix that is characteristic of Caché global names, for example: `dataGlobal="^myTree"` Using *dataGlobal* excludes *OnGetNodeInfo*. For details, see the section "<dynaTree> Using a Global" in this chapter.

Attribute	Description
imageContracted	File name for the image to display when the <dynaTree> is contracted. The user clicks on this image to expand the group. The default for *imageContracted* is the plus sign ▸ whose file name is `contracted.gif`. You may specify an alternate image. The image file that you identify must reside in the `images` directory under the Caché installation directory. *imageContracted* applies only when *showLines* is false. When *showLines* is true, a plus sign automatically appears as part of the graphical image that displays the lines.
imageExpanded	File name for the image to display when the <dynaTree> is expanded. The user clicks on this image to contract the group. The default for *imageExpanded* is the minus sign ▾ whose file name is `expanded.gif`. You may specify an alternate image. The image file that you identify must reside in the `images` directory under the Caché installation directory. *imageExpanded* applies only when *showLines* is false. When *showLines* is true, a minus sign automatically appears as part of the graphical image that displays the lines.
imageFolderClosed	File name for the image to display beside a closed folder node. This is a node that has children but is currently closed (contracted). The user clicks on this image to expand the node. The default for *imageFolderClosed* is a closed folder icon  whose file name is `folderclosed.gif`. You may specify an alternate image. The image file that you identify must reside in the `images` directory under the Caché installation directory.
imageFolderOpen	File name for the image to display beside an open folder node. This is a node that has children and is currently open (expanded). The user clicks on this image to contract the node. The default for *imageFolderOpen* is an open folder icon  whose file name is `folderopen.gif`. You may specify an alternate image. The image file that you identify must reside in the `images` directory under the Caché installation directory.
imageNode	File name for the image to display beside a leaf node. This is a node with no children in the <dynaTree>. The user clicks on this image to select the node. The default for *imageNode* is a file icon  whose file name is `node.gif`. You may specify an alternate image. The image file that you identify must reside in the `images` directory under the Caché installation directory.
showLines	If true, show dashed lines between the nodes of the tree. The default is false. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
onchange	The *onchange* event handler for the <dynaTree>. Zen invokes this handler when the current value of the <dynaTree> changes. See "Zen Component Event Handlers."
onclick	Client-side JavaScript expression that Zen invokes when the user clicks on a node within the <dynaTree>.

Attribute	Description
ondblclick	Client-side JavaScript expression that Zen invokes when the user double-clicks on a node within the <dynaTree>.
OnGetNodeInfo	Name of a server-side callback method that reports information about each node within the tree. Using *OnGetNodeInfo* excludes *dataGlobal*. For details, see the section "<dynaTree> Using an OnGetNodeInfo Callback" in this chapter.
OnGetTreeInfo	Name of a server-side callback method that fills in a local array that supplies the contents of the tree. Using *OnGetTreeInfo* excludes both *OnGetNodeInfo* and *dataGlobal*. For details, see the section "<dynaTree> Using an OnGetTreeInfo Callback" in this chapter.
selectedIndex	0-based index of the currently selected node in the <dynaTree>. The default *selectedIndex* is –1 (nothing is selected).

When you work with %ZEN.Component.dynaTree programmatically, you must also know about the following properties of the dynaTree class:

- Each <parameter> element provided in the original <dynaTree> definition in XData Contents becomes a member of the dynaTree parameters property, a list collection of %ZEN.Auxiliary.parameter objects. Each <parameter> acquires an ordinal position in the parameters collection: 1, 2, 3, etc.
- The read-only text property holds the text (display) value of the currently selected node within the tree. This is the node label that displays in the <dynaTree>.
- The read-only value property holds the logical (actual) value of the currently selected node within the tree. This is the string that is activated when the user clicks the corresponding label in the <dynaTree>.

7.6.2.6 <dynaTree> Drag and Drop

<dynaTree> supports data drag operations when its *dragEnabled* attribute is set to true. This is possible because each node in a <dynaTree> has a logical value and a display value defined.

Data drag is explained in the "Data Drag and Drop" section in the chapter "Zen Controls." Essentially, the user clicks the mouse button down while the cursor is positioned on a <dynaTree> node, then moves the mouse away from the <dynaTree> while still holding down the button. Data drag captures the current value of the <dynaTree> node where the drag operation began. If the logical value and displayed value are different, this difference is preserved when the data is captured.

The data acquired from a <dynaTree> node can be dropped on any Zen component that has drop enabled. Generally this is a Zen control. <dynaTree> does not support data drop, as it is not a control and cannot accept data input from the client side.

7.7 Filters

Zen offers a <buttonView> component that allows you to lay out a set of navigation choices as buttons in a grid. Typically, <buttonView> is used to display progressive filter buttons that allow a user to narrow down a search, as on the InterSystems documentation home page:

In the previous example, the user has just clicked the **JavaScript** button. This has disabled all the other buttons in the same category as **JavaScript**, and has caused the **Reset** button to activate and change color. To re-enable all the buttons in this category, the user could click the **JavaScript** button again.

A <buttonView> may contain multiple categories of buttons. Each category works like a set of radio buttons, in that only one button in the category may be selected at any time. In the following example, many more categories in the <buttonView> have been selected.

Every <buttonView> component automatically provides a **Reset** button that is enabled only when the user has made at least one selection. **Reset** appears as the top left entry in the grid. If clicked, **Reset** cancels all current user selections in the <buttonView>.

7.7.1 <buttonView>

The <buttonView> component does not contain any other components. It provides its buttons dynamically, via a server-side callback method. <buttonView> has the following attributes:

Attribute	Description
Zen component attributes	<buttonView> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
columns	Number of columns for the grid. If not specified, the default is 4. The number of rows is determined dynamically depending on the total number of items and the *columns* value.
OnGetButtonInfo	Name of a server-side callback method that provides the list of items to display for this component. For details, see the "<buttonView> OnGetButtonInfo" section following this table.

Attribute	Description
onselect	The *onselect* event handler for the <buttonView>. Zen invokes this handler when the user selects a new button. The *onselect* handler method must have one input parameter that you can use to pass it the current string *value* of the <buttonView>. For details, see the "<buttonView> onselect" section following this table. See "Zen Component Event Handlers."
seed	Allows you to pass some arbitrary value to the *OnGetButtonInfo* callback. The *seed* value can be a literal string, or it can contain a Zen #()# runtime expression. For suggestions about how to use the <buttonView> *seed* value, see the "<buttonView> OnGetButtonInfo" section following this table.
value	A string containing the currently selected value from the component. Each entry in the <buttonView> *value* string takes the following form: *category*:*button*; If there are multiple categories in the <buttonView>, and the user makes a selection in more than one category, Zen appends each subsequent choice to the existing *value* string. When the user has selected multiple buttons, the *value* string takes the following form: *category*:*button*;*category*:*button*;*category*:*button*; The user can click the built-in **Reset** button to empty the *value* string and start over.

7.7.1.1 <buttonView> XData Contents

The following XData Contents excerpt defines a <buttonView> and an <html> component:

XML

```
<hgroup width="100%" cellVAlign="top" cellAlign="left">
  <buttonView id="filter" columns="2"
              OnGetButtonInfo="GetFilterList"
              onselect="zenPage.filterChange(value);"
              />
  <spacer width="10"/>
  <html id="myContent" OnDrawContent="DrawMyContent"/>
</hgroup>
```

The next several topics explain how user selections in this <buttonView> can cause this <html> component to invoke its *OnDrawContent* method, effectively redrawing the <html> component based on the user's selections in the <buttonView>.

7.7.1.2 <buttonView> OnGetButtonInfo

The server-side callback method identified by the <buttonView> *OnGetButtonInfo* attribute must be defined in the page class. It must return a %Status value, and it must have two input parameters in the following order:

- A %String. If you provide a *seed* value for the <buttonView>, it is automatically passed to the *OnGetButtonInfo* callback method.
- An array passed by reference. This array contains an array of button descriptions when the method returns.

To support the previous <buttonView> syntax example, the page class would also need to define a method called **GetFilterList** with the following signature. Zen automatically invokes the method and uses the array that it creates, whenever it displays the <buttonView> component:

```
ClassMethod GetFilterList(pSeed As %String, ByRef pInfo) As %Status
```

The *OnGetButtonInfo* callback method must fill the array that was passed to it by reference with a required *category*, *caption*, and *value* for each button. Optionally, each array entry can also contain *tooltip* text and a *disabled* flag for the button. An ObjectScript statement such as the following would correctly set the array entry for button number *n*:

`Set pInfo(`*n*`) = $LB(`*category*`,`*caption*`,`*button*`,`*tooltip*`,`*disabled*`)`

Where:

- $LB is shorthand for the ObjectScript function $LISTBUILD.
- *n* is the 1–based button number.
- *category* is the name assigned to the category that contains this button.

  You must group buttons in the <buttonView> into categories. All buttons with the same *category* value work together, like a set of radio buttons. When one button in a *category* is selected, none of the other buttons in that *category* can be selected.

  This feature is visible in the figure at the beginning of this topic, for which all of the buttons that correspond to programming languages have had their *category* set to the same string. The user has clicked one of these buttons (**JavaScript**) so all of the other buttons in this *category* are disabled. This action has not affected any of the buttons in other categories.
- *caption* is the text that appears on the face of the button. You can use a $$$Text macro to support localization of the *caption*.
- *button* is part of the string value that is recorded when this button is selected. Each entry in the *value* string for the <buttonView> component takes the following form:

  category`:`*button*`;`

  If the user makes a selection in more than one category, Zen appends each subsequent choice to the existing *value* string, as in:

  category`:`*button*`;`*category*`:`*button*`;`*category*`:`*button*`;`
- *tooltip* is the text that displays when the user hovers the cursor over the button. The *tooltip* can also use $$$Text.
- *disabled*, if true, disables this button. The button still displays, but is grayed out.

The following is a sample ObjectScript statement that defines a button for a <buttonView> by assigning a value to an entry in the *pInfo* array:

```
Set pInfo($I(pInfo))=
$LB("adv",$$$Text("Expert Tools"),"Expert",$$$Text("For expert users only"))
```

Where $I is shorthand for the ObjectScript function $INCREMENT.

7.7.1.3 <buttonView> onselect

The example in "<buttonView> XData Contents" identifies an *onselect* handler called **filterChange** that accepts one parameter, the current *value* of the <buttonView> component:

XML

```
<buttonView id="filter" columns="2"
              OnGetButtonInfo="GetFilterList"
              onselect="zenPage.filterChange(value);"
              />
```

filterChange could then work as follows:

Class Member

```
ClientMethod filterChange(list) [ Language = javascript ]
{
  var html = zen('myContent');
  var div = html.getEnclosingDiv();
  div.scrollTop = 0;
  html.seed = list;
  html.refreshContents();
}
```

This *onselect* handler does the following:

1. Calls the JavaScript function **zen** to retrieve the <html> component that appears on the page along with the <buttonView>. This <html> component had the following definition in XData Contents:

   ### XML

   ```
   <html id="myContent" OnDrawContent="DrawMyContent"/>
   ```

   That is why **filterChange** uses this statement:

   ```
   var html = zen('myContent');
   ```

2. Assigns the incoming <buttonView> *value* to the <html> *seed* attribute:

   ```
   html.seed = list;
   ```

3. Refreshes the <html> component:

   ```
   html.refreshContents();
   ```

   This automatically invokes the *OnDrawContent* handler for the <html> component. This callback must be a server-side method defined in the page class. It must accept a %String as input and return a %Status data type. In this case, its name is **DrawMyContent**.

   DrawMyContent uses the <html> *seed* (that is, the <buttonView> *value*) to determine what to do while redrawing the <html> component. It is up to this method to figure out how to use the filtering information provided by multiple user selections in the <buttonView>. The <buttonView> simply provides a *value*.

8

Popup Windows and Dialogs

This chapter describes Zen components that "pop up" to display their contents over the main application page. The tradeoff for the flexibility provided by these components is that you must provide more programming in the page class to make them effective. This chapter describes the following components:

- "Modal Groups" — Make HTML content visible in response to a user event.
- "Popup Windows" — Pop up a new window that displays a Zen page or any other content.
- "Dialogs" — Pop up a dialog that prompts the user and accepts user input.

8.1 Modal Groups

A *modal group* is a specialized group component that normally does not display its contents. When the modal group is made visible, only its contents receive user events. Any event that occurs outside of the modal group (such as a mouse click) automatically hides the modal group. Zen uses this modal mechanism when it creates drop-down menus and drop-down combobox controls. You can also use this mechanism for popup items.

You can define the contents of a modal group either by placing a <modalGroup> component within the page class or by creating an instance of the %ZEN.Component.modalGroup class programmatically. There are three options:

- "Static" — Place a <modalGroup> component in the page class XData Contents block. Its contents remain hidden until a client-side method on the page calls the modalGroup **show** method.
- "Dynamic" — Have the page call its **createComponent** method to create a modalGroup component dynamically. Add components to the group. Display it by calling the modalGroup **show** method.
- "Built-in" — Have the page call the modalGroup **show** method to display one of the built-in modal groups: `"msgBox"` or `"calendar"`

To close the current modal group, the page must invoke its **endModal** method. Generally **endModal** is triggered by an event handler for one of the buttons in the modal group (OK, Cancel, or similar). It is an error to call **endModal** if there is no modal group currently displayed.

While a modal group has the editing focus, keyboard controls are disabled. For example, it is not possible to use **Tab** to move from field to field within the modal group. This prevents the user from (inadvertently) pressing **Tab** to navigate through all the fields and then back to the page, while the modal group has focus. The modal group keeps the focus until the user either clicks away from it or clicks the button that has been set up to close the modal group.

Note: For Internet Explorer (IE) only: When displaying a modal group, Zen hides the contents of any <embed> elements (that is, SVG content). This is because IE ignores zindex setting for <embed> elements.

You can customize the behavior of a component within a modal group by modifying its modal callback methods:

- **onStartModalHandler**(*zindex*) — If present, fires upon notification that this component is about to become modal. That is, it is pushed to the front of the display. The caller supplies a *zindex* value large enough to ensure that this component is placed above all others currently visible in the browser.
- **onEndModalHandler**(*zindex*) — If present, fires upon notification that this component is about to stop being modal. That is, it is no longer be pushed to the front of the display. The caller supplies a *zindex* value small enough to ensure that this component returns to its normal layer relative to other components in the browser.

8.1.1 Static Modal Groups

In static mode, the modal group is defined within a page class XData Contents block using XML statements. The contents of the group are hidden until the modal group component's **show** method is called; then they are displayed in a popup window.

The following steps set this up in the page class:

1. Supply a <modalGroup> definition within the <page> in XData Contents. For example:

   XML

   ```
   <modalGroup id="mgStatic" groupTitle="Popup">
     <text id="mgText" label="Value:" />
     <button id="mgButton" caption="OK" onclick="zenPage.mgBtnClick();"/>
   </modalGroup>
   ```

2. Provide a client-side method to display the modal group. For example:

   Class Member

   ```
   ClientMethod modalGroupStatic() [ Language = javascript ]
   {
     var group = this.getComponentById('mgStatic');
     group.show();
   }
   ```

   This method:

   - Finds the <modalGroup> on the <page> by calling **getComponentById** with the <modalGroup> *id* value
   - Calls the client-side **show** method. No arguments are needed because the <modalGroup> definition provides all the necessary information to **show**.

3. Somewhere within the <page> you must give the user a mechanism to invoke the **modalGroupStatic** method to display the modal group. The following example provides a button:

   XML

   ```
   <button caption="Enter a New Value"
           onclick="zenPage.modalGroupStatic();"
           title="Display a modal group using a static definition." />
   ```

4. For testing purposes, you may provide the <page> with a field in which to echo the data from the popup:

   XML

   ```
   <html id="mgHtml">No data entered yet. </html>
   ```

The page XData Contents block now looks like this.

XML

```
<page xmlns="http://www.intersystems.com/zen"
      xmlns:demo="http://www.intersystems.com/zendemo" height="100%">
  <html id="mgHtml">No data entered yet. </html>
  <button caption="Enter a New Value"
          onclick="zenPage.modalGroupStatic();"
          title="Display a modal group using a static definition." />
  <modalGroup id="mgStatic" groupTitle="Popup">
    <text id="mgText" label="Value:" />
    <button id="mgButton" caption="OK" onclick="zenPage.mgBtnClick();"/>
  </modalGroup>
</page>
```

5. You must give the user a mechanism to close the popup. Additionally, if your popup invited the user to enter values, you want to retrieve and use these values. The following client-side method in the page class does this:

Class Member

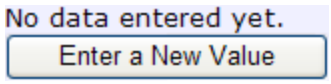
```
ClientMethod mgBtnClick() [ Language = javascript ]
{
  // get value from text field
  var ctrl = zen('mgText');

  // write user value into HTML component
  zenSetProp('mgHtml','content','User entered: ' + ctrl.getValue());

  // hide the modal group
  zenPage.endModal();
}
```

This method:

- Executes when the user clicks OK in the popup. This works because step 1 identified this method as the *onclick* event handler for the <button> in the <modalGroup>.
- Finds the <text> control from the <modalGroup> by calling the JavaScript function **zen** with the <text> *id* value
- Gets the value entered into the <text> control by calling **getValue**
- Finds the <html> component on the <page> and sets the content property of the <html> component, by calling the JavaScript utility function zenSetProp() with the <html> *id* value. The new <html> content value concatenates a literal string with the newly acquired <text> value.
- Calls the page's **endModal** method to close the popup.

The user may interact with this sample page as follows:

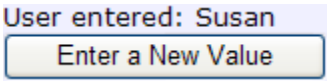
1. The user initially sees:



2. Clicking the button on this page invokes the **modalGroupStatic** method. This causes a window to pop up in front of the main page. The contents of the popup are the contents of the <modalGroup>. The popup title is the <modalGroup> *groupTitle* text.

   In the illustration below, the user has typed a value in the <text> control within the popup:

3. Clicking the OK button in this popup invokes the **mgBtnClick** method. This closes the popup and changes the contents of the <html> component on the <page> to echo the user input, as follows:



The above example is similar to one available in the class ZENDemo.MethodTest in the SAMPLES namespace. You can try this sample from the Zen Demo main page by choosing **Home**, **Overview**, **Methods**, then **Modal Group: Static**. To get started, see "The Zen Demo" as described in the "Introducing Zen" chapter of *Using Zen*.

8.1.2 Dynamic Modal Groups

In dynamic mode, the page creates a modal group component programmatically, by calling the client-side JavaScript methods **createComponent**, **addChild**, and **setProperty**. The sequence culminates in a call to the modalGroup **show** method with the appropriate arguments.

The following steps set this up in the page class:

1. Provide a client-side method to display the modal group. For example:

Class Member

```
ClientMethod modalGroupDynamic() [ Language = javascript ]
{
  // create a modal group
  var group = this.createComponent('modalGroup');

  // add components dynamically
  var col = this.createComponent('colorPicker');
  group.addChild(col);
  col.setProperty('id','myCol');

  var radio = this.createComponent('radioSet');
  radio.setProperty('id','myRadio');
  group.addChild(radio);

  zenPage.addChild(group);
  zenPage.refreshContents(true);
  radio.setProperty('valueList','elm,maple,oak');

  var btn = this.createComponent('button');
  group.addChild(btn);
  btn.setProperty('caption','Save');
  btn.setProperty('onclick','zenPage.btnClick();');

  // Show the group in "dynamic" mode.
  zenPage.refreshContents(true);
  group.show();
}
```

This method:

- Calls the page's **createComponent** method to add a modalGroup component
- Adds a colorPicker to the group
  - Calls the page's **createComponent** method to create a colorPicker control
  - Calls the modal group's **addChild** method to add the colorPicker to the group

  - Calls the color picker's **setProperty** method to give the colorPicker an *id* value

- Adds a radioSet to the group in the same way
- Adds a button to the group in the same way
- Calls the button's **setProperty** method to set its *onclick* value to the name of a client-side method that executes when the user clicks this button
- Calls the modal group's **show** method with these arguments:
  - A title for the popup window
  - The keyword `'dynamic'`
  - The keyword `null` in argument positions three, four, and five
  - An optional width value of `'236'`

  For argument details, see "The show Method" in this chapter.

2. Somewhere within the <page> you must give the user a mechanism to invoke the **modalGroupDynamic** method to display the modal group. The following example provides a button:

   XML

   ```
   <button caption="Choose Plantings"
           onclick="zenPage.modalGroupDynamic();"
           title="Display a modal group using a dynamic definition." />
   ```

3. For testing purposes, you may provide the <page> with a field in which to echo the data from the popup:

   XML

   ```
   <html id="mgHtml">No data entered yet. </html>
   ```

   The page XData Contents block now looks like this.

   XML

   ```
   <page xmlns="http://www.intersystems.com/zen"
         xmlns:demo="http://www.intersystems.com/zendemo" height="100%">
     <html id="mgHtml">No data entered yet. </html>
     <button caption="Choose Plantings"
             onclick="zenPage.modalGroupDynamic();"
             title="Display a modal group using a dynamic definition." />
   </page>
   ```

4. You must give the user a mechanism to close the popup. Additionally, if your popup invited the user to enter values, you want to retrieve and use these values. The following client-side method in the page class does this:

   Class Member

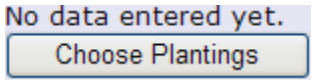
   ```
   ClientMethod btnClick() [ Language = javascript ]
   {
     // get values from controls
     var col = zen('myCol');
     var radio = zen('myRadio');

     // write user values into HTML component
     zenSetProp('mgHtml','content','User entered: ' +
                       col.getValue() + ' ' + radio.getValue());

     // hide the modal group
     zenPage.endModal();
   }
   ```

This method:

- Executes when the user clicks Save in the popup. This works because step 1 identified this method as the *onclick* event handler for the Save button in the modal group.
- Finds the color picker and radio set controls by calling the JavaScript utility function zen() with their *id* values.
- Finds the <html> component on the <page> and sets the content property of the <html> component by calling the JavaScript utility function zenSetProp() with the <html> *id* value. The new <html> content concatenates a literal string with the color picker and radio set values, acquired using **getValue**().
- Calls the page's **endModal**() method to close the popup.

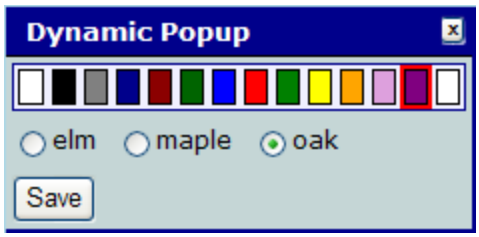
The user may interact with this sample page as follows:

1. The user initially sees:

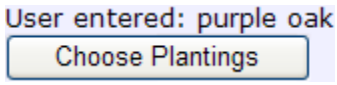


2. Clicking the button on this page invokes the **modalGroupDynamic** method. This causes a window to pop up in front of the main page. The contents of the popup are the child components that **modalGroupDynamic** added. The popup title is the first argument that **modalGroupDynamic** passed to the **show** method.

   In the illustration below, the user has clicked on a color and a radio button:



3. Clicking the Save button in this popup invokes the **btnClick** method. This closes the popup and changes the contents of the <html> component on the <page> as follows:



8.1.3 Built-in Modal Groups

In built-in mode, Zen dynamically creates and displays one of its built-in modal groups. This option is much simpler than the steps for a dynamic modal group. There are two options for built-in modal groups:

- `"calendar"` — Display the built-in "calendar box."
- `"msgBox"` — Display the built-in "message box."

8.1.3.1 Calendar

The following steps add a calendar popup to a Zen page class:

1. Provide a client-side method to display the modal group.

Class Member

```
ClientMethod modalGroupCalendar() [ Language = javascript ]
{
  var group = zenPage.createComponent('modalGroup');
  group.setProperty('onaction','zenPage.calendarAction(group);');
  group.show('Select a date:','calendar','2005-12-12');
}
```

This method:

- Calls the page's **createComponent** method to add a modalGroup component
- Calls the modal group's **setProperty** method that sets its *onaction* value to the name of a client-side method. **setProperty** executes when the user performs any action on this popup, such as selecting a date value from the calendar. It accepts an argument, `group`, that represents the modalGroup instance.
- Calls the modal group's **show** method with three arguments:
  - A title for the popup window
  - The keyword `'calendar'`
  - An optional pre-selected date for the calendar to display when it pops up

2. Somewhere within the <page> you must give the user a mechanism to invoke the method that displays the modal group. The following example provides a button:

XML

```
<button caption="Display a Calendar"
        onclick="zenPage.modalGroupCalendar();" />
```

3. Finally, you must retrieve and use the date value acquired by the calendar control, and then close the popup. The following client-side method in the page class does this:

Class Member

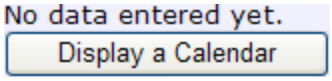
```
ClientMethod calendarAction(group) [ Language = javascript ]
{
  alert("You selected: " + group.getValue());

  // write user value into HTML component
  zenSetProp('mgHtml','content','User entered: ' + group.getValue());
}
```

This method:

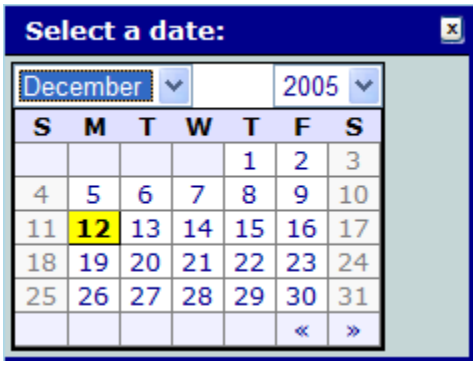
- Executes when the user clicks on a specific date value in the popup. This works because step 1 identified this method as the *onaction* event handler for the modal group.
- Calls the modal group's **getValue** method to retrieve the date value entered into the calendar.
- Issues a JavaScript alert message that confirms the value.
- Finds the <html> component on the <page> and sets the content property of the <html> component by calling the JavaScript utility function zenSetProp() with the <html> *id* value. The new <html> content value concatenates a literal string with the newly acquired date value.
- *Does not call* the page's **endModal** method. The built-in calendar modal group closes automatically when the user chooses a date.

The user may interact with this sample page as follows:

1. The user initially sees:

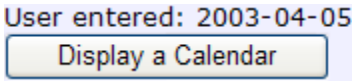


2. Clicking the button on this page invokes the **modalGroupCalendar** method. This causes a window to pop up in front of the main page. The popup title is the first argument that **modalGroupCalendar** passed to the **show** method. The currently selected date is the one specified by the third **show** argument.



3. Selecting a different month, year, and date from the preselected value automatically invokes the **calendarAction** method. This closes the popup and displays a browser alert message that echoes the new date value:

4. Dismissing the alert makes it easy to see that the contents of the <html> component on the <page> have now changed as follows:



The above example is similar to one available in the class ZENDemo.MethodTest in the SAMPLES namespace. You can try this sample from the Zen Demo main page by choosing **Home**, **Overview**, **Methods**, and **Modal Group: Calendar**. To get started, see "The Zen Demo" as described in the "Introducing Zen" chapter of *Using Zen*.

8.1.3.2 Message Box

The following steps add a message box popup to a Zen page class:

1. Provide a client-side method to display the modal group.

Class Member

```
ClientMethod modalGroupMsg() [ Language = javascript ]
{
  var group = this.createComponent('modalGroup');
  group.show('My New Message','msgBox',
  'This<br>message<br>contains <span style="color: red">HTML</span>!');
}
```

This method:

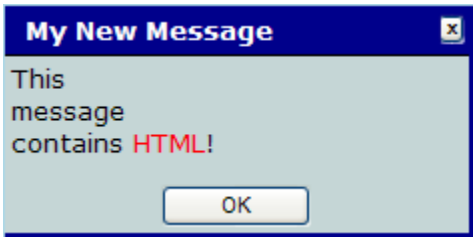
- Calls the page's **createComponent** method to add a modalGroup component
- Calls the modal group's **show** method with three arguments:
  - A title for the popup window
  - The keyword `'msgBox'`
  - HTML-formatted message content for the popup message

2. Somewhere within the <page> you must give the user a mechanism to invoke the method that displays the modal group. The following example provide a button:

   XML

   ```
   <button caption="Display a Message"
           onclick="zenPage.modalGroupMsg();" />
   ```

3. When the user clicks this button, the popup displays:

   Clicking OK on the popup dismisses it.

The above example is similar to one available in the class ZENDemo.MethodTest in the SAMPLES namespace. You can try this sample from the Zen Demo main page **Home**, **Overview**, **Methods**, and **Modal Group: MsgBox**. To get started, see "The Zen Demo" as described in the "Introducing Zen" chapter of *Using Zen*.

8.1.4 The show Method

show is a client-side JavaScript method that displays a modal group. It has no return value, and up to eight optional arguments. Previous topics in the "Modal Groups" section have discussed some of these arguments. A complete list follows:

- *title* — Title for the popup window. For static modal groups, this overrides the *groupTitle* value in the <modalGroup> definition.
- *type* — One of the following keywords:
  - `"calendar"` — Display the built-in calendar box.
  - `"dynamic"` — Display a dynamically created modal group.
  - `"msgBox"` — Display the built-in message box.
  - `"static"` — Display a statically defined <modalGroup>.

  If no *type* argument is supplied, the default is `"static"` if a <modalGroup> definition exists within the <page>. Otherwise the default is `"dynamic"`.
- *value* — Value to display when a built-in modal group is used. When the *type* is:
  - `"calendar"` — *value* is a date in YYYY-MM-DD format. When the window pops up, the calendar displays this month and year with *value* as a pre-selected date.

  - `"msgBox"` — *value* is the HTML-formatted message content for the popup message.

- *top* — Vertical coordinate of the top left corner of the popup window (0 is the top of the screen)
- *left* — Horizontal coordinate of the top left corner of the popup window (0 is the far left of the screen)
- *wid* — Width of the popup window
- *hgt* — Height of the popup window
- *parms* — Object containing a set of additional characteristics passed on to the modalGroup as a set of name-value pairs. Typically this offers a way to pass additional parameters to the popup calendar.

8.1.5 <modalGroup> Attributes

The <modalGroup> component is the XML projection of the %ZEN.Component.modalGroup class. The following table lists the <modalGroup> attributes that are available when defining a <modalGroup> within a page class XData Contents definition.

Attribute	Description
Zen group attributes	<modalGroup> has the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
groupTitle	Title to display at the top of the modal group. For static modal groups, you can set the *groupTitle* value in the <modalGroup> definition. Otherwise, this value is set dynamically by the first argument of the **show** method. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
okCaption	Caption displayed in OK button for a message box. The default is "OK". Although you can enter ordinary text for this attribute, it has the underlying data type %%ZEN.Datatype.caption. See "Zen Attribute Data Types."
onaction	The *onaction* event event handler for the <modalGroup>. Zen invokes this handler whenever the user takes action on a built-in modal group popup (`"msgBox"` or `"calendar"`). See "Zen Component Event Handlers." If you provide a *seed* value for the <modalGroup>, it is automatically passed to the *onaction* event handler.
onhideGroup	Client-side JavaScript expression that runs when the modal group is hidden.
onshowGroup	Client-side JavaScript expression that runs when the modal group is made visible.
seed	Allows you to pass some arbitrary value to the *onaction* event handler.
value	(Read-only) *value* is set by the **show** method; applications should not set this. However, it is sometimes useful to retrieve this value, for example when working with the built-in calendar modal group.

8.2 Popup Windows

A *popup window* is a new browser window that pops up over the currently active window in response to a user event. The popup window becomes the active window as soon as it is displayed, and remains dominant until the user closes it. The following table lists the client-side JavaScript methods that control popup windows. Except for **onPopupAction**, these Zen page methods are final and cannot be overridden. **onPopupAction** is a Zen component method that needs to be overridden to control popup behavior. Since Zen pages are also components, they support the **onPopupAction** client-side method.

By default, Zen renders popup windows as elements drawn on the current Zen page. If you want popup dialogs to display in separate windows, you can set the global `^%ISC.ZEN.useSoftModals` to 0 or override the method **%OnUseSoftModals** so that it returns 0.

Note that you should avoid using characters other than alphanumeric characters and the underscore in pop-up window names because doing so is incompatible with certain browsers

Table 8–1: Client Side Methods for Controlling Popup Windows

Client-Side Method	Description
launchPopupWindow(...)	Launch a popup window. Either identify the parent component, or allow it to default to the current Zen page.
firePopupAction(...)	Notify the parent component for the current popup that a user action has occurred. Optionally close the popup as well.
onPopupAction(...)	The parent component invokes this method each time it is notified that a user action has occurred on the popup.
cancelPopup()	Close the current popup with no further action.

To display a popup window from a Zen page, follow these steps:

1. Create a Zen page class to display as a popup window. In our examples, this is “the popup page” or:

   ```
   MyApp.MyDialog.cls
   ```

   When you design a popup page, add properties to it that correspond to URI parameters you expect to pass to it when invoking the popup; for example:

   ```
   Property ABC As %String(ZENURL="ABC");
   ```

2. Open a Zen page class that you plan to use as the window that launches the popup. In our examples, this is “the parent page.”

3. Add a client-side method to the parent page and make this method call **launchPopupWindow**.

   In the following example, the popup page class name appears as the first argument in the call to **launchPopupWindow**, correctly enclosed in `zenLink` syntax with single quotes. This call to **launchPopupWindow** passes a single URI parameter called ABC to the popup page with a value of 22:

Class Member

```
ClientMethod showPopupWindow() [ Language = javascript ]
{
  var parms = new Object();
  parms.ABC = 22;
  zenPage.launchPopupWindow(
    zenLink('MyApp.MyDialog.cls'),
    'A True Dialogue',
    'status,scrollbars,resizable,width=400,height=300',
    parms);
}
```

For more detailed descriptions, see the list of arguments following these instructions.

4. Place a component on the parent page whose *onclick* attribute invokes the client-side method that invokes **launchPopupWindow**. For example:

XML

```
<button caption="Initiate Conversation"
        onclick="zenPage.showPopupWindow();"
        title="A demonstration of launching a popup window" />
```

5. Examine the popup page class. Make sure it provides a component that invokes the **firePopupAction** method. **firePopupAction** notifies the parent of the popup window that a user action has occurred in the popup. **firePopupAction** has three arguments:

   - *action* — If an *action* parameter is not specified, the literal string `ok` is the action code.
   - *value* — Your code must provide a *value*; this *value* is automatically passed as an argument to the parent's **onPopupAction** method.
   - *close* — *close* is true by default. If true, the popup window closes after notifying the parent window.

   For example:

```
// tell our parent window that OK was pressed; do not close this window.
this.firePopupAction('apply',this.getDialogValue(),false);
```

6. In the parent component class, override the **onPopupAction** method to provide an appropriate response when the popup page fires a popup action. The general idea is to retrieve and use the *value* returned by the popup page. If you did not identify a parent component, the parent is the page that invoked the popup. If the parent is the page, you must override **onPopupAction** in the parent page class. In the **onPopupAction** method signature:

   - *popupName* is the name of the popup window sending the action.
   - *action* is the name of the action.
   - *value* is the value associated with the action.

   All of these arguments are automatically passed to **onPopupAction** from the popup page's **firePopupAction** method.

   Within the **onPopupAction** method, you may use the returned *value* in any way you wish. The following example saves the value into the *content* property of an <html> component:

   Important: When Internet Explorer is the browser, do not use a submit operation to save values from any popup window. Simply save the values.

Class Member

```
ClientMethod onPopupAction(popupName, action, value) [ Language = javascript ]
{
  zenSetProp('mgHtml','content','User entered: ' + value);
}
```

7. If you created a button or link on the parent page, such as the one shown in step 4 above, clicking that button now displays your popup page in a separate window, overlaying the parent page.

The client side method **launchPopupWindow** has the following arguments, in sequential order:

1. *url* – A string that identifies the content you wish to display, either:
   - The name of a Zen page class, including its package name and .cls extension
   - The URI of the desired content

   The JavaScript function **zenLink** makes sure that any additional URI parameters, such as session tracking values, are included in the link. You may use **zenLink** with the class name or with any URI. In the following example, the client-side method uses JavaScript string concatenation to construct a URI from several disparate elements before making the call to the Zen page method **launchPopupWindow**. The example also uses the **escape** function to correctly format any variable values used in the string:

Class Member

```
ClientMethod editCSSValue(context) [ Language = javascript ]
{
  var ctrl = this.getComponentById('value');
  var value = ctrl.getValue();
  var url = zenLink('%ZEN.Dialog.cssDeclarationEditor.cls?context='
            + escape(context) + '&declaration=' + escape(value) );
  zenPage.launchPopupWindow(url,
            'CSSDeclarationEditor',
            'resizable,width=500,height=700' );
}
```

2. *pageName* – A string that identifies the popup window.
3. features – A comma-separated list of window attributes expected by the JavaScript function **window_open**.

   The results of these attributes are browser-dependent. Some developers prefer the Internet Explorer behavior, which is truly modal: the modal window stays on top and nothing can happen in the application until the user dismisses the modal window. Other developers prefer the non-modal behavior, which other browsers use regardless of how you set up this call: the modal window is just another window in the application and other windows can take priority over the modal window if the user clicks on them.

   If you want to make the behavior in Internet Explorer consistent with the behavior in other browsers, include the string `modal=no` in the list of window attributes for **launchPopupWindow**. For example:

```
zenPage.launchPopupWindow(url,
          'CSSDeclarationEditor',
          'resizable,width=500,height=700,modal=no'
          );
```

4. *parms* – If provided, *parms* is an array of values to be used as parameters in the URI string that invokes the popup window. You can use the *parms* argument to supply parameters for the URI so that you do not have to worry about correctly formatting and concatenating URI parameters in the *url* argument.

The convention for populating the *parms* array with values is as follows:

```
var parms = new Object();
parms.Aaa = 10;
parms.Bbb = 20;
zenPage.launchPopupWindow(url,
          'CSSDeclarationEditor',
          'resizable,width=500,height=700,modal=no'
          parms);
```

Given the above sample code, **launchPopupWindow** uses *Aaa* and *Bbb* as URI parameters, and gives these parameters the values that you assigned to the corresponding members of the *parms* array (10 and 20). **launchPopupWindow** takes care of all the details of correctly escaping these parameter values and combining them with the *url* value in the URI string.

Each member of the *parms* array must correspond to a ZENURL property in the popup page class; for example:

```
Property Aaa As %String(ZENURL="Aaa");
Property Bbb As %String(ZENURL="Bbb");
```

5. *parent* – If provided, *parent* identifies the Zen component to be notified when the **firePopupAction** method fires. This can be any component on the page, or the page object itself. If no *parent* value is supplied, the parent defaults to the page. If provided, the *parent* value is the system-assigned index number used internally to identify this component. For components within repeating groups, *parent* includes a dot followed by a number indicating the (1–based) position of this component within the group. Applications can use but should not set the *parent* value.

8.3 Dialogs

Zen includes some built-in classes designed to work as popup dialog windows. A Zen dialog window can contain any of the usual Zen components. All dialog windows are subclasses of %ZEN.Dialog.standardDialog, which is a Zen page class. This base class provides classic dialog buttons such as OK, Apply, and Cancel, as well as conventions for retrieving the values entered into the dialog if the user clicks OK.

This topic describes the conventions for working with Zen dialog windows:

- Displaying the built-in file select window
- Displaying the built-in color select window
- Displaying an search dialog window
- Creating a new dialog window based on %ZEN.Dialog.standardDialog
- Creating a new dialog window template, as an alternative to %ZEN.Dialog.standardDialog

8.3.1 File Selection Dialog Window

The %ZEN.Dialog.fileSelect window displays and lets the user choose from a list of directories and files on the server system. The resulting value is a full path and filename.

Important: To use the fileSelect dialog to view the file system on the server, the current user must hold USE privileges on one of the following resources: %Admin_Manage, %Admin_Operate, %Admin_Security, or %Development.

To display the fileSelect dialog as a popup window, use the steps in the "Popup Windows" section of this chapter, but in the step that describes adding a client-side method, provide a method similar to the following:

Class Member

```
ClientMethod showPopupFile() [ Language = javascript ]
{
var pathname = '\Temp';
var showdir = '0';
zenPage.launchPopupWindow(
  '%ZEN.Dialog.fileSelect.cls?Dir=' + escape(pathname) +
  '&showdirectoryonly=' + showdir,
  'FileSelectWindow',
  'status,scrollbars,resizable,width=660,height=700');
}
```

The method calls the Zen page method **launchPopupWindow** with these arguments:

- URI string beginning with the filename of the %ZEN.Dialog.fileSelect class. The filename is all that is needed, but in this case, the URI continues with various parameters, including:
  - Pathname for the file search window. This is either a full pathname or a path relative to the Caché system management directory (`\mgr` under the Caché installation directory). If omitted, the default is the Caché system management directory.
  - A `showdirectoryonly` value of 0 so that files are visible in the display. You can omit this, as 0 is the default. Set `showdirectoryonly` to 1 if you want to show directories only.
- Identifier for the popup window
- Comma-separated list of popup window characteristics

For a more complete example using %ZEN.Dialog.fileSelect, see ZENTest.FormTest2 in the SAMPLES namespace. For further details about the **launchPopupWindow** method, see the "Popup Windows" section, earlier in this chapter.

8.3.2 Color Selection Dialog Window

The %ZEN.Dialog.colorSelect window lets the user choose from a set of colored cells. The resulting value is an HTML color value, such as #F3FDDA. To display the fileSelect dialog as a popup window, use the steps in the "Popup Windows" section of this chapter, but in step 2, provide a client-side method similar to the following:

Class Member

```
ClientMethod showPopupColor() [ Language = javascript ]
{
  zenPage.launchPopupWindow(
    '%ZEN.Dialog.colorSelect.cls',
    'ColorPicker',
    'status,scrollbars,resizable,width=500,height=700');}
```

This method calls the client-side function **launchPopupWindow** with arguments as follows:

- Filename of the %ZEN.Dialog.colorSelect class
- Identifier for the popup window
- Comma-separated list of popup window characteristics

For further details about the **launchPopupWindow** method, see the "Popup Windows" section, earlier in this chapter.

8.3.3 Search Dialog Window

The %ZEN.Dialog.searchDialog window lets the user construct and execute an SQL query. This class has the following properties:

- *query* — the SQL statement used to populate the search form. This value cannot be passed in as a ZENURL parameter. Instead, applications should subclass %ZEN.Dialog.searchDialog and provide the *query* value using server-side logic in the subclass.
- *paramNames* — provides a comma-separated list of names to display for parameters in the search form.

8.3.4 Creating a Dialog Window

To create a new type of Zen dialog window based on %ZEN.Dialog.standardDialog:

1. Subclass %ZEN.Dialog.standardDialog.
2. Add a new XData block with:
   - The name `dialogBody`
   - A <pane> as the container
   - An *id* value for the component that you identify as the source of user input

   For example:

   Class Member

   ```
   XData dialogBody
   {
     <pane>
       <text label="I say: " value="What do you think?" />
       <textarea label="And you say: " id="yourReply" />
     </pane>
   }
   ```

3. Override the server-side callback method **%OnGetTitle**, for example:

   Class Member

   ```
   Method %OnGetTitle() As %String
   {
     Quit $$$TextHTML("This is My Dialog")
   }
   ```

4. Override the server-side callback method **%OnGetSubtitle**, for example:

   Class Member

   ```
   Method %OnGetSubtitle() As %String
   {
     Quit $$$TextHTML("by John Smith")
   }
   ```

5. Override the client-side JavaScript method **getDialogValue** so that it returns a value, for example:

   Class Member

   ```
   ClientMethod getDialogValue() [ Language = javascript ]
   {
     return this.getComponentById('yourReply').getValue()
   }
   ```

6. To give the dialog an Apply button (in addition to the automatic OK and Cancel buttons) override the APPLYBUTTON parameter and set it to 1:

Class Member

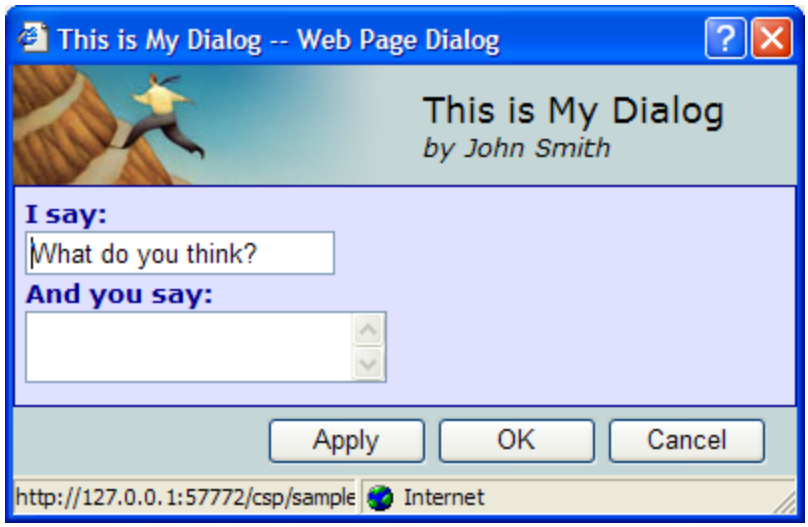
```
Parameter APPLYBUTTON = 1;
```

7. To enable the $$$Text localization macros, give the DOMAIN parameter a meaningful value:

Class Member

```
Parameter DOMAIN = "MyDomain";
```

8. When displayed, this dialog appears as follows:



8.3.5 Creating a Dialog Window Template

Suppose you want to apply your own styling to the dialogs, rather than using the same layout and graphics as in the Zen standard dialog. The most straightforward way to accomplish this is to create a new dialog window template class and use it as an alternative to %ZEN.Dialog.standardDialog in the above instructions. The steps are:

1. Subclass %ZEN.Dialog.standardDialog.
2. In your subclass, apply the style differences you require.
3. When creating new dialogs using the instructions in the "Creating a Dialog Window" section in this chapter, extend the new subclass instead of extending %ZEN.Dialog.standardDialog
4. When you need a file selector class:
   - Copy the class %ZEN.Dialog.fileSelect
   - Edit the copy so that it extends your new dialog window template class instead of extending %ZEN.Dialog.standardDialog
5. When you need a color selector class:
   - Copy the class %ZEN.Dialog.colorSelect
   - Edit the copy so that it extends your new dialog window template class instead of extending %ZEN.Dialog.standardDialog

9

Other Zen Components

This chapter describes components that fit into none of the categories defined in other chapters of this book: layout, tables, meters, charts, forms, controls, navigation aids, or popups.

- "<colorPane>" — A large, detailed palette from which the user can select colors
- "<colorWheel>" — A continuous color gradient from which the user can select colors.
- "<dynaView>" — Display a view box based on data provided by a callback method.
- "<fieldSet>" — A group component that helps visually organize a form
- "<finderPane>" — A Finder-like browser for hierarchically organized data.
- "<html>" — A snippet of arbitrary HTML content
- "<iframe>" — A static image or other content within a frame
- "<repeatingGroup>" — Repeat the same layout for each item returned by a runtime query.
- "<schedulePane>" — Display a daily, weekly, or monthly calendar with time slots for each date.
- "<timer>" — A non-visible component that fires an event after a time interval

If you have examined the full roster of components, including those listed above, and still do not see the functionality you need, consider adding a custom component to the Zen library. For full instructions, see the "Custom Components" chapter in *Developing Zen Applications*.

9.1 HTML Content

The Zen <html> component provides a way to place arbitrary HTML content within a Zen page. The HTML content is displayed within the Zen page exactly as specified. This content is drawn within an enclosing HTML in the same manner as any other Zen component.

There are several ways to define the content displayed by the <html> element:

- Literally specify the content within the <html> component:

  XML

```
<html id="myContent">
This is some <b>HTML</b>.
</html>
```

The content must be *well-formed*. That is, any HTML elements within the content must have matching closing elements, and any attribute values must be quoted correctly. The <script> tag is not allowed.

The content may not reference other Zen components. However, it may contain Zen #()# runtime expressions.

- Define a server-side **OnDrawContent** callback method that writes HTML content. If you define such a method, and reference it from the <html> element, this method is called whenever the <html> component is displayed.

  XML

  ```
  <html id="myContent" OnDrawContent="GetHTMLContent" />
  ```

  The method itself would look something like this:

  Class Member

  ```
  Method GetHTMLContent(pSeed As %String) As %Status
  {
      &html<This is some HTML!>
      Quit $$$OK
  }
  ```

- The <html> component has a content property that is a string of HTML statements. You can set the content property's initial value by placing the desired HTML tags in between the <html> and </html> elements in XData Contents, or by referencing an *OnDrawContent* callback.

  Client code can change the value of the content property by resetting it from JavaScript. The following example uses the JavaScript utility function zenSetProp() to do this:

  ```
  // get html content
  zenSetProp('myContent','content','some new content');
  ```

The <html> component has the following attributes:

Attribute	Description
Zen component attributes	<html> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
OnDrawContent	Name of a server-side callback method in the Zen page class. This method provides HTML content using &html<> syntax or WRITE commands. Zen invokes this method whenever it draws the <html> component, automatically passing it a %String that contains the component's *seed* value. The callback must return a %Status data type. The following is a valid method signature: `Method DrawMe(pSeed As %String) As %Status` To use the above method as the callback, the developer would set `OnDrawContent="DrawMe"` for the <html> component.
seed	Allows you to pass some arbitrary value to the *OnDrawContent* callback.

9.2 Framed Content

The Zen <iframe> component is a wrapper for the HTML <iframe> element. It may display images or other content within a frame.

9.2.1 <iframe> Attributes

The Zen <iframe> component has the following attributes:

Attribute	Description
Zen component attributes	<iframe> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
longdesc	URI that provides the link to the long (text) description of the frame.
frameAlign	The *align* value for the HTML <iframe>: `"left"`, `"right"`, or `"center"`.
frameBorder	The *frameborder* value for the HTML <iframe>. If true, the frame has a border; if false it does not have a border. This attribute has the underlying data type %ZEN.Datatype.boolean. See "Zen Attribute Data Types."
scrolling	The *scrolling* value for the HTML <iframe>: `"auto"`, `"yes"`, or `"no"`.
src	Identifies the pathname and filename of the frame content. The path is either an absolute pathname or a path relative to the Caché installation directory, for example: `<iframe id="myFrame" src="/csp/myApp/myPic.png" />`

If you wish to change the contents of the frame from the client, you can do so programmatically by resetting the value of the src property of the iframe object.

When changing the src property value on the client side, do not simply set the property, as in:

```
var frame=zen('contentFrame');
frame.src='C:\myfile';
```

Instead, use the setProperty() method, as in:

```
var frame=zen('contentFrame');
frame.setProperty('src','C:\myfile');
```

9.2.2 Images as Button Controls

If you wish to place an image on a button control, so that you can define an *onclick* event or other types of event for it, use the <image> control, which gives greater control over event handling. For details, see the description of the <image> control in the chapter "Zen Controls."

9.2.3 Rendering Image Data Streams

InterSystems advises that you do not use <iframe> to render image data streams. Instead, use <image> as described in the chapter "Zen Controls." This convention is imperative when the image is in JPG format and the browser is Internet Explorer (IE). This topic explains the details.

Suppose you want to use an <iframe> to render content retrieved from the database via a CSP routine that serves up a data stream. The stream could be PDFs, GIFs, JPGs, Docs, RTFs, etc. It turns out that, when the image is in JPG format and the browser is IE, rendering JPG data in an <iframe> incapacitates core Zen functionality.

The root of the problem is in the way IE renders data streams.

When browsers like Firefox see that the data stream isn't an HTML page, they create a dummy DOM to wrap around the incoming stream; GIFs and JPGs become HTML <IMG> tags with their *src* attribute set to the URI provided, PDFs automatically generate HTML <EMBED> tags to launch the viewer, etc. In these cases, a DOM is always created and the normal scoping rules of JavaScript execution apply.

IE is different. It does not create a DOM; it swaps out normal page processing in favor of a dedicated viewer for the content in question. For ordinary web pages this is usually not an issue. If you are just loading a picture or downloading a PDF, there is nothing to interact with the DOM, so it does not matter if the DOM is there or not.

The problem arises when the content is being loaded into an <iframe> under IE.

Content requiring an external plug-in, such a PDFs or RTFs, seem to work fine with IE. The embedded GIF viewer also seems to play well. The JPG viewer, however, does not. If a JPG image is loaded as a direct stream into an <iframe>, when IE initializes the embedded viewer it obliterates, not just the JavaScript execution space for the <iframe>, but the *entire* JavaScript namespace for the page, deleting global client-side variables and JavaScript function definitions. After the viewer is initialized, JavaScript continues to function outside the <iframe> for event callbacks and such, but chances are good that any client-side data initialized before the loading of the <iframe> has been reset to null.

The best cross-platform solution for dealing with cases like these is to use the Zen components <iframe> and <image> as follows:

- <iframe> for HTML documents or data streams requiring external plug-ins (such as PDFs)
- <image> to render data streams of types GIF, JPG or PNG

The only downside to this solution is that if streams are being retrieved from the database, two queries are required: one to determine the type of data being sent and a second to send the actual data. The overhead in this case is almost exclusively the network overhead, as the data volume is trivial. Therefore, the best practice for rendering data streams retrieved from the database is as follows:

1. Define the template display page with both an <iframe> and an <image>. Both components are initially hidden (`hidden='true'`).
2. Query to find out what type of data is to be rendered.
3. Based on this query, set the *src* attribute of either the <iframe> or the <image> to the URI of the data stream and set the corresponding component to be visible (`hidden='false'`)
4. When the page is closing, reset the displayed component to be hidden (`hidden='true'`) so that the page is ready for the next data stream.

Other permutations are possible, but this is the basic idea. It is not necessary to use an <iframe> to display image data on a web page, and doing so with a JPG under IE breaks Zen, so do not use <iframe> for this purpose; use <image> instead.

9.3 Timer

The <timer> component has no visual representation. It raises a client-side event after a specified time period. You may place a <timer> on the Zen page within XData Contents as follows:

Class Member

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
<page>
  <timer ontimeout="zenPage.timeout(zenThis);" timeout="10000" />
</page>
}
```

<timer> has the following attributes:

Attribute	Description
ontimeout	The *ontimeout* event handler for the timer. Zen invokes this handler whenever the timer runs out. It must accept an argument, *timer*, that represents this timer object. The <timer> element can use the built-in variable `zenThis` to pass the timer object to the method. See "Zen Component Event Handlers."
timeout	Number of milliseconds in the timer. Zen automatically starts the timer when the page is first loaded. When the *timeout* time has elapsed, Zen fires the *ontimeout* expression. The example above sets a *timeout* value of 10,000 milliseconds. This provides an timeout value of 10 seconds.

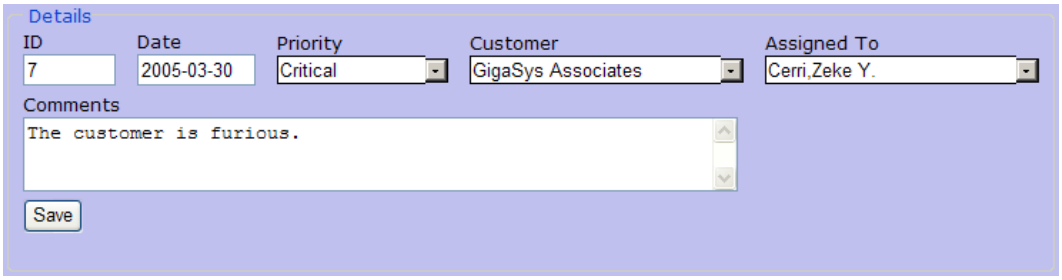
The client-side *ontimeout* method might look something like the following. A <timer> only fires its *ontimeout* event once, so your JavaScript method must restart the timer (by calling its **startTimer** method) before exiting. The following example does this:

Class Member

```
ClientMethod timeout(timer) [ Language = javascript ]
{
  // ...do something

  // restart the timer
  timer.startTimer();
}
```

9.4 Field Sets

A <fieldSet> is a group component that draws an outer box around the children and displays a title within this box. This creates a visual panel that can help to organize a page. The following example from the SAMPLES namespace class ZENApp.HelpDesk defines a panel called "Details" that contains a form with several different controls.



The <fieldSet> that produces the above example looks like this:

XML

```
<fieldSet id="detailGroup" legend="Details">
  <form id="detailForm" layout="vertical" labelPosition="top"
        cellStyle="padding: 2px; padding-left: 5px; padding-right: 5px;"
        onchange="zenPage.detailFormChange(zenThis);" >
    <hgroup>
      <text id="ID" name="ID" label="ID" readOnly="true" size="5"/>
      <spacer width="15"/>
      <text id="CreateDate" name="CreateDate" label="Date"
            readOnly="true" size="8"/>
      <spacer width="15"/>
      <dataCombo id="Priority" name="Priority" label="Priority" size="12"
                 dropdownHeight="150px" editable="false" unrestricted="true"
                 sql="SELECT Name FROM ZENApp_Data.Priority ORDER BY Name"/>
      <spacer width="15"/>
      <dataCombo id="Customer" name="Customer" label="Customer" size="24"
                 dropdownHeight="150px" editable="false" unrestricted="true"
                 sql="SELECT ID,Name FROM ZENApp_Data.Customer ORDER BY Name"/>
      <spacer width="15"/>
      <dataCombo id="AssignedTo" name="AssignedTo" label="Assigned To" size="24"
                 dropdownHeight="150px" editable="false" unrestricted="true"
                 sql="SELECT ID,Name FROM ZENApp_Data.Employee ORDER BY Name"/>
    </hgroup>
    <textarea id="Comments" name="Comments"
              label="Comments" rows="3" cols="60"/>
    <button id="btnSave" caption="Save" disabled="true"
            onclick="zenPage.detailFormSave();" />
  </form>
</fieldSet>
```

A <fieldSet> group may contain a <form>, as above, or a <form> may contain a <fieldSet>. A <fieldSet> provides a *visual* grouping only; it is not a Zen form, because it does not provide any form behavior such as validation or submit. <fieldSet> has the following attributes:

Attribute	Description
Zen group attributes	<fieldSet> has the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
legend	Text specifying the caption to display for this <fieldSet>. The *legend* can be a literal string, or it can contain a Zen #()# runtime expression. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
title	Text specifying a popup message to display for this <fieldSet>. The *title* can be a literal string, or it can contain a Zen #()# runtime expression. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."

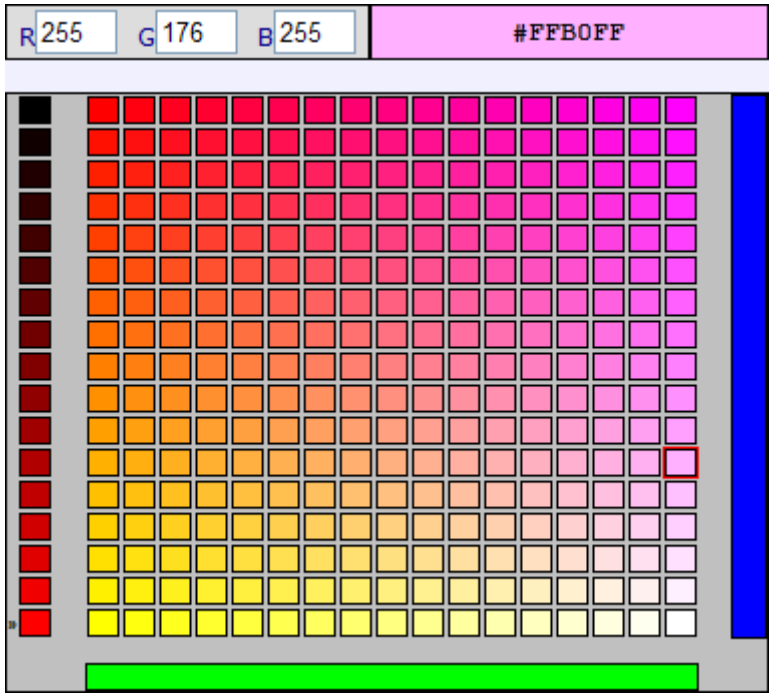
9.5 Color Selector

The <colorPane> component lets the user view and select colors or enter RGB values in a large color palette. The palette is a grid that can be visualized as one possible slice from a three-dimensional cube of available colors. The <colorPane> captures a slice from the cube by accepting the user's choice of one of three faces (red, green, or blue) and slicing through the cube, parallel to that face, at some saturation level (brighter or dimmer) to produce a grid of colors.

From the user's viewpoint, the <colorPane> has these features:

- The user selects a color by clicking a square in the palette.
- The user can cycle through different faces and slices of the color cube by clicking on the red, green, or blue color bars at left, bottom, and right. The segmented bar at left allows the user to select a slice that uses a brighter or dimmer range of color combinations.
- The bar at top right displays the currently selected color and its corresponding HTML hexadecimal value. The user can copy (but not edit) the text value from this color bar.
- At top left, the user can change the text entry fields **R**, **G**, and **B** to any number in the range 0–255, then click on the color bar at top right to apply these changes and see the result. The text fields **R**, **G**, and **B** also respond to user selections from the palette.

A <colorPane> with a color selected looks like this:



<colorPane> has the following attributes. For a simpler color selection component, see "<colorPicker>."

Attribute	Description
Zen component attributes	<colorPane> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen* <colorPane> is not a full-blown control component, but it does allow the user to select a value, and it can be used within Zen forms. Any control attribute not listed in this table is not available for <colorPane>.
currCol	0–based column number of the currently selected cell in the color palette. The default is 0. The range is from 0 to 15.
currRow	0–based row number of the currently selected cell in the color palette. The default is 0. The range is from 0 to 15.
currSlice	0–based slice number of the currently selected slice of the 3–dimensional color cube. The default is 0. The range is from 0 to 15.
face	Face number of the currently selected face of the 3–dimensional color cube: 1, 2, or 3. The default is 1.
onchange	The *onchange* event handler for the <colorPane>. Zen invokes this handler when the *value* of the <colorPane> component changes. See "Zen Component Event Handlers."
ondblclick	Client-side JavaScript expression that Zen invokes when the user double-clicks on the <colorPane> component.
value	String that identifies the most recently selected HTML hexadecimal value. The default is: #FFFFFF

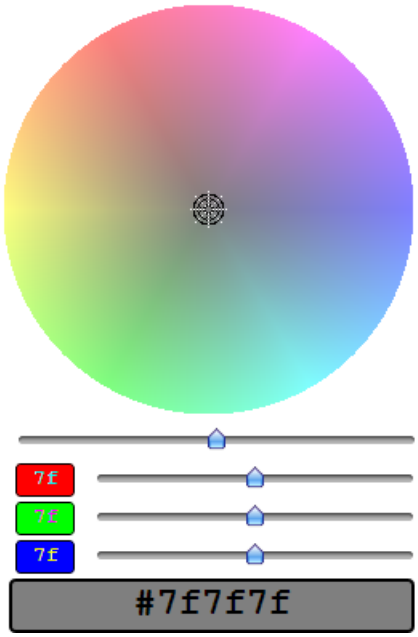
9.6 Color Wheel

The <colorWheel> component lets the user view and select colors from a continuous color gradient. The color wheel actually provides two different groups of controls. You can either use the color wheel and intensity slider to pick a shade from the wheel, or use the separate RGB sliders to mix a specific color.

When you use the disk and intensity slider, move the slider to the left to see darker tones, or to the right to see lighter ones. The rendering of the disk offers real time feedback as to the current settings. Once you have the right range of colors displayed, click on the desired region of the disk. A cross-hair shows you where you clicked and the background of the preview box at the bottom of the widget updates to reflect your choice. Also, the RGB sliders update to reflect the selected color.

When you use the RGB sliders, drag the sliders to adjust the contributions of the individual RGB color guns. The current value for each gun displays in the box to the left of the slider. The color resulting from the contributions of all three guns appears in the background of the preview box at the bottom of the control. Also, the intensity slider, color disk, and cross-hair location update to reflect the current color.

A <colorWheel> with a color selected looks like this:



<colorWheel> has the following attributes. For a color selection component that lets you select from a color palette, see "<colorPane>.". For a simple color selection component that lets you choose from a limited set of colors, see "<colorPicker>."

Attribute	Description
Zen component attributes	<colorWheel> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
showPreview	Show the preview box at the bottom of the control that is filled with the currently selected color and stamped with the HTML Hex color specification.
showRGBPanel	Show the panel containing three sliders that allows the user to select values for the RGB color guns.
value	The value that is automatically initialized for the <colorWheel> when it is displayed. This property does not hold the value selected by the user. You need to use the *getValue* method to get the current value of a control. The default value is: `#7f7f7f`

9.7 Repeating Group

A <repeatingGroup> is a specialized group component that defines the layout for a single group item, then outputs multiple items with this layout. The number of items, and the data contained in each item, is determined by a runtime query.

<repeatingGroup> has the following attributes:

Attribute	Description
Zen group attributes	<repeatingGroup> has the same style and layout attributes as any Zen group. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*. The default *layout* for a <repeatingGroup> is vertical.
Data source attributes	<repeatingGroup> has similar attributes to <tablePane> for specifying the data source. <repeatingGroup> supports *maxRows*, *queryClass*, *queryName*, and *sql*. For details and examples, see the following sections in the chapter "Zen Tables": • How to use query attributes with <repeatingGroup>: – "Data Sources" (*maxRows*) – "Specifying an SQL Query " (*sql*) – "Referencing a Class Query" (*queryClass*, *queryName*) • How to provide <parameter> elements within <repeatingGroup>: – "Query Parameters"
onclickitem	The *onclickitem* event handler for the <repeatingGroup>. Zen invokes this handler when the user clicks on an item within the repeating group. See "Zen Component Event Handlers."
selectedIndex	The (0–based) index number of the currently selected item. The default value is –1, which indicates there is no current selection.

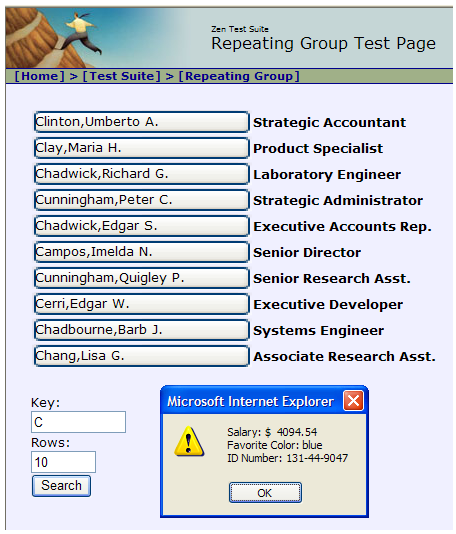
The typical layout strategy for a repeating group is for the <repeatingGroup> element to enclose a horizontal group that contains the individual item. Since the default layout for a <repeatingGroup> is vertical, the result of this arrangement is that each item is laid out horizontally, while the set of items is laid out from top to bottom.

The following excerpt from an XData Contents block provides an example of this layout strategy for <repeatingGroup>. This is similar to the XData Contents block in the class ZENTest.RepeatingGroupTest in the SAMPLES namespace. Note the <hgroup> that defines the individual entry. This <hgroup> contains a <button> and an <html> component.

XML

```
<vgroup>
  <repeatingGroup id="repeatingGroup"
                  maxRows="1000"
                  sql="SELECT TOP ? Name,Title,SSN,Salary,FavoriteColor
                  FROM ZENDemo_Data.Employee WHERE Name %STARTSWITH ?">
    <parameter value="#(%page.Rows)#" />
    <parameter value="#(%page.SearchKey)#" />
    <hgroup>
      <button caption="#(%query.Name)#"
              title="Salary: $  #(%query.Salary)#"
              onclick="alert('Salary: $  #(%query.Salary)#\n
                              Favorite Color: #(%query.FavoriteColor)#\n
                              ID Number: #(%query.SSN)#')" />
      <html>
        <b>#(%query.Title)#</b>
      </html>
    </hgroup>
  </repeatingGroup>
  <spacer height="25"/>
  <form>
    <text id="search" label="Key:" value="#(%page.SearchKey)#" size="10"/>
    <text id="rows" label="Rows:" value="#(%page.Rows)#" size="5"/>
    <button caption="Search" onclick="zenPage.refreshGroup();" />
  </form>
</vgroup>
```

The following figure illustrates sample output from this excerpt. To produce this output, the user entered the key C, requested 10 rows, and then clicked **Search**. This caused Zen to display the repeating group using the first 10 entries that start with the letter C. The user then clicked an employee name in the group to display the alert window, which contains data retrieved from the server during the query.



On the output side, only one object is created for each child of the repeating group, but there are multiple HTML renderings within that object, one for each child. The rendered HTML incorporates the current item number into each generated *id* value used to identify the HTML elements on the page.

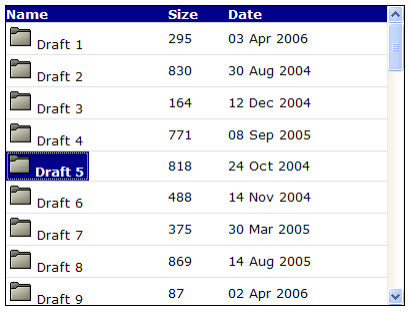
InterSystems recommends that you keep the use of <repeatingGroups> reasonably simple. Repeating groups within repeating groups are not supported.

9.8 Dynamic View

The <dynaView> component displays a set of user-defined items within a view box, similar to the way a file dialog box might display a set of files or directories. <dynaView> items can displayed in one of two modes:

- "list" — All items are displayed compactly in a grid. Only the first column of data is displayed.
- "details" — Each item is displayed as one row within a table. Each row may present several columns of data.

A <dynaView> in “details” mode looks like the following example, based on the class ZENTest.DynaViewTest in the SAMPLES namespace.



9.8.1 <dynaView> OnGetViewContents Callback Method

<dynaView> gets its data by invoking a server-side callback method defined in the Zen page class. The method name is specified using the <dynaView> *OnGetViewContents* attribute, as shown in the following example. In this example, the method name is **GetViewContents**.

XML

```
<dynaView id="view" viewType="details"
          OnGetViewContents="GetViewContents"
          onchange="zenPage.viewChange(zenThis);" >
  <parameter paramName="label" value="Draft" />
</dynaView>
```

A <dynaView> can contain zero or more <parameter> elements. Each <parameter> specifies an input parameter for the *OnGetViewContents* callback method. Each <parameter> may have the following attributes:

Attribute	Description

id	If you wish to be able to access the parameter to change its value at runtime, in response to user actions on the client side, then you must give the <parameter> element an *id*. With a parameter defined as follows:
	
	XML
	
	`<parameter id="parmLabel" paramName="label" value="Draft" />`
	
	A client-side JavaScript method in the same Zen page class could change the value of this parameter to “Final” as follows:
	
	`var parm = zen(parmLabel); parm.value = "Final";`

paramName	The *paramName* must be unique within the <dynaView>. It becomes a subscript in the array of parameters passed to the callback method.
value	The *value* supplied for a <parameter> can be a literal string, or it can contain a Zen #()# runtime expression.

The *OnGetViewContents* callback method must have a signature that looks like this:

```
Method GetViewContents(
    ByRef pParms As %String,
    Output pContents As %String,
    Output pHeaders As %String) As %Status
```

Where:

- The method returns a %Status value indicating success or failure.
- *pParms* represents any <parameter> elements defined by the <dynaView>. *pParms* is an array. Each member of this array uses its *paramName* as a subscript and its *value* as a value.
- *pContents* is a multidimensional array that describes the set of items that is displayed within the view box. The array is subscripted by item number (starting with 1). Each array element is a $List structure containing the following information. You can use the ObjectScript function $LISTBUILD ($LB) to create each list item for *pContents*, using statements such as the following inside the callback method:

  ```
  Set pContents(n) =
  $LB(textValue,logicalValue,icon,col2,col3)
  ```

Where:

- *textValue* is the value displayed for the item
- *logicalValue* is a logical value associated with the item
- *icon* is the URI of the image to be displayed with the item (if any)
- Any subsequent values (*col2*, *col3*, and so forth) define the values to display in any additional columns when the dynaView is in "details" mode

- *pHeaders* is a multidimensional array that provides a set of column headers, subscripted by column number. When the dynaView is in "details" mode, this is used to define how many columns of data to display and what the column headers are. For example:

```
// define 3 column headers
Set pHeaders(1) = "Name"
Set pHeaders(2) = "Size"
Set pHeaders(3) = "Date"
```

The icons used in a dynaView belong to a CSS class called dynaViewIcon, which enables you to use CSS to control the appearance of the icon. For example, the following CSS insures that the icon is displayed as 16px x 16px:

```
.dynaViewIcon {
    height: 16px;
    width: 16px;
}
```

9.8.2 <dynaView> Attributes

The previous topic described the *OnGetViewContents* attribute in detail. The following table lists all the <dynaView> attributes.

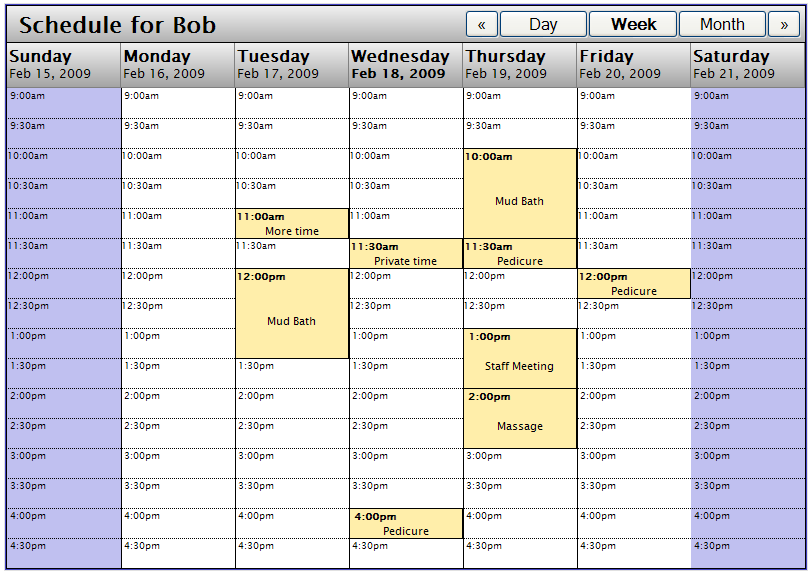
Property	Description
onchange	The *onchange* event handler for the <dynaView>. Zen invokes this handler when the current value of the <dynaView> changes. See "Zen Component Event Handlers."
onclick	Client-side JavaScript expression that Zen invokes when the user clicks on a node within the <dynaView>.
ondblclick	Client-side JavaScript expression that Zen invokes when the user double-clicks on a node within the <dynaView>.
OnGetViewContents	Name of a server-side callback method that provides the contents of the <dynaView>. For details, see the section "<dynaView> OnGetViewContents Callback Method."
onselect	Client-side JavaScript expression that Zen invokes when the user selects an item by either pressing the **Enter** key or double-clicking
rows	Number of rows to display when the <dynaView> is in list mode.
selectedIndex	0-based index of the currently selected node in the <dynaView>. The default *selectedIndex* is –1 (nothing is selected).
viewType	Specifies how the contents of the <dynaView> should be displayed. Possible values are: • "list" — all items are displayed compactly in rows and columns • "details" — each item is displayed as one row within a table

When you work with %ZEN.Component.dynaView programmatically, you must also know about the following properties of the dynaView class:

- Each <parameter> element provided in the original <dynaView> definition in XData Contents becomes a member of the dynaView parameters property, a list collection of %ZEN.Auxiliary.parameter objects. Each <parameter> acquires an ordinal position in the parameters collection: 1, 2, 3, etc.
- The read-only text property holds the text (display) value of the currently selected node within the tree. This is the node label that displays in the <dynaView>.
- The read-only value property holds the logical (actual) value of the currently selected node within the tree. This is the string that is activated when the user clicks the corresponding label in the <dynaView>.

9.9 Schedule Calendar

A <schedulePane> displays a daily, weekly, or monthly calendar with time slots for each date. Users can define appointments and place them in the appropriate time slots. The following example shows a <schedulePane> in Week mode with several appointments defined. This example is similar to the class ZENTest.SchedulePaneTest in the SAMPLES namespace.



Note: The <schedulePane> is not a control, and does not return a value. For simple date selection controls that enable users to enter dates as values in forms, see the "Dates" section in the chapter "Zen Controls."

9.9.1 <schedulePane> OnGetScheduleInfo Callback Method

<schedulePane> gets its data by invoking a server-side callback method defined in the Zen page class. The method name is specified using the <schedulePane> *OnGetScheduleInfo* attribute, as shown in the following example. In this example, the method name is **GetScheduleInfo**.

XML

```
<schedulePane id="schedule" caption="Schedule for Bob"
  dateFormat="5" interval="30" startTime="540" endTime="1020"
  dropEnabled="true" ondrop="zenPage.scheduleDataDrop(dragData);"
  onselectitem="zenPage.selectItem(id,time);"
  OnGetScheduleInfo="GetScheduleInfo">
  <parameter id="parmPerson" paramName="Person" value="Bob" />
</schedulePane>
```

A <schedulePane> can contain zero or more <parameter> elements. Each <parameter> specifies an input parameter for the *OnGetScheduleInfo* callback method. Each <parameter> may have the following attributes:

Attribute	Description

id	If you wish to be able to access the parameter to change its value at runtime, in response to user actions on the client side, then you must give the <parameter> element an *id*. With a parameter defined as follows:
	
	`<parameter id="parmPerson" paramName="Person" value="Bob" />`
	
	A client-side JavaScript method in the same Zen page class could change the value of this parameter to "Sally" as follows:
	
	`var parm = zen('parmPerson'); parm.value = "Sally";`

paramName	The *paramName* must be unique within the <schedulePane>. It becomes a subscript in the array of parameters passed to the callback method.
value	The *value* supplied for a <parameter> can be a literal string, or it can contain a Zen #()# runtime expression.

The *OnGetScheduleInfo* callback method must have a signature that looks like this:

```
ClassMethod GetScheduleInfo
  (ByRef pParms As %String,
   pStartDate As %Date,
   pEndDate As %Date,
   ByRef pInfo As %List) As %Boolean
```

Where:

- The method returns a %Boolean value which is true to indicate success; otherwise it is false.
- *pParms* represents any <parameter> elements defined by the <schedulePane>. *pParms* is an array. Each member of this array uses its *paramName* as a subscript and its *value* as a value.
- *pStartDate* is the starting date and time in $HOROLOG ($H) format, the internal storage format for dates in Caché. As described in the *Caché ObjectScript Reference*, this format gives the number of days since December 31, 1840; then a comma; then the number of seconds since midnight. That is:

  date,*time*

  Where both *date* and *time* are integers.
- *pEndDate* is the ending date and time in $HOROLOG ($H) format.
- *pInfo* is a multidimensional array that describes the time slots that are displayed within the schedule calendar. The array is subscripted as follows:
  - *day* is the date in $HOROLOG ($H) format.

- *time* is an integer expressing the start time for this time slot as a number of minutes since midnight. Your code is more readable if you express this value as a calculated multiple of 60, using `9*60` for 9 a.m. instead of the literal value `540`, but either is correct.
- *n* is an integer (starting with 1, and usually equal to 1) that you can use as needed to distinguish between time slots that have the same *day* and *time* values.

Each *pInfo* array element is a $List structure containing the following information. You can use the ObjectScript function $LISTBUILD ($LB) to create each list item for *pInfo*, using statements such as the following inside the callback method:

```
Set pInfo(day,time,n) =
$LISTBUILD(duration,id,text,type,style)
```

Where:

- *duration* is an integer specifying the number of minutes to reserve for this time slot in the schedule.
- *id* is an integer that the *onselectitem* event handler uses to uniquely identify this time slot when it is selected in the schedule.
- *text* is the text description to display in this time slot in the schedule.
- *type* is either 0 or 1:
  - 0 is an unavailable time slot
  - 1 is a scheduled time slot
- *style* is a CSS style to apply to this time slot in the schedule. This value overrides any competing styles that might otherwise apply to this time slot.

The following example sets up a 60-minute staff meeting at 9 a.m. on the given start date, designates it as a scheduled time slot, and colors its background green on the schedule calendar display:

Class Member

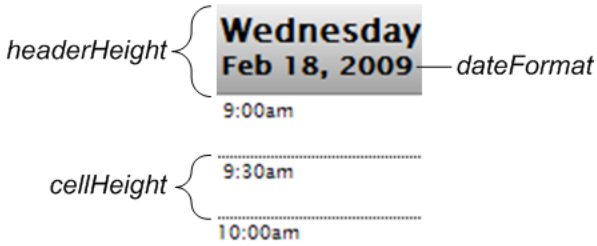
```
ClassMethod GetScheduleInfo(ByRef pParms As %String,
                            pStartDate As %Date,
                            pEndDate As %Date,
                            ByRef pInfo As %List)
                            As %Boolean
{
  Set pInfo(pStartDate,9*60,1)
  = $LB(60,1,"Staff Meeting",1,"background: green;")
  Quit 1
}
```

You can also provide a style that applies to all of the time slots in a particular day by setting a top node of the *pInfo* array to a CSS style value. For example:

```
  Set pInfo(day) = "background: yellow;"
```

9.9.2 <schedulePane> Attributes

This topic has already described the *OnGetScheduleInfo* attribute in detail. The following table lists all the <schedulePane> attributes.

Attribute	Description
Zen component attributes	<schedulePane> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
caption	Text to display as a caption along the top of the schedule. The text is *not* HTML escaped, so it can contain markup. The *caption* value can be a literal string, or it can contain a Zen #()# runtime expression. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
cellHeight	Integer giving the height of time slots in the schedule calendar, in pixels. The default is 30, as shown in the illustration for *dateFormat*. Do not specify units with *cellHeight*.
date	Date to display, in YYYY-MM-DD format. The default is the current date. The schedule calendar displays a range of days that contain this date value. The actual range depends on the current value of the *view* attribute: `"day"`, `"week"`, or `"month"`. The *date* value can be a literal string, or it can contain a Zen #()# runtime expression.
dateFormat	Integer that specifies the date format to use in the headers for the schedule calendar. The <schedulePane> automatically displays this date below the name of the day of the week. The default *dateFormat* is –1. This generates the date in the format "Feb 18, 2009" as shown in the following illustration. As the illustration shows, you can also configure the *headerHeight* and *cellHeight* attributes for this part of the schedule calendar display.  For a list of the integers you can use as values for *dateFormat*, see the description of *dformat* in the "Parameters" section for the $ZDATETIME ($ZDT) function in the *Caché ObjectScript Reference*. *dateFormat* allows nearly the same range of values as *dformat*. *dateFormat* supports –1 and the values 1 through 13.

Attribute	Description
dayList	Comma-separated list of the full names of days of the week. The default *dayList* value uses the $$$Text macro with the English values Sunday through Saturday. Using this macro ensures that this *dayList* value is automatically included in the list of strings to translate for your application when you export them from the message dictionary, as long as you also set the DOMAIN parameter in the Zen class. For details, see the "$$$Text Macros" section in the "Zen Localization" chapter of *Developing Zen Applications*. You can use $$$Text with *dayList* because its ZENLOCALIZE datatype parameter is set to 1 (true). Any localized *dayList* string must remain a comma-separated list of seven items when translated. Be sure to coordinate *dayList* changes with *shortDayList* and *firstDayOfWeek*.
endTime	Ending time for the daily time slots to be displayed on the schedule calendar. This is an integer expressing the number of minutes since midnight. The default *endTime* is 1080, which means 6 p.m. The range for *endTime* is 0 through 1440, which covers all 24 hours in the day. If the end user schedules an appointment that falls outside the *startTime* and *endTime*, the calendar expands to display that time slot. The *endTime* value can be a literal string, or it can contain a Zen #()# runtime expression.
firstDayOfWeek	Integer that specifies which day of the week is displayed as the starting day of the week in the schedule calendar. 0 means Sunday, 1 means Monday, and so on up to 6, which means Saturday. The default *firstDayOfWeek* is 0. The purpose of the *firstDayOfWeek* attribute is to allow you to customize the schedule calendar for different locales. Be sure to coordinate *firstDayOfWeek* changes with *dayList* and *shortDayList*.
headerHeight	Integer giving the height of the header row at the top of the schedule calendar, in pixels. This header row is where Zen displays the names of days and their dates. The default is 40, as shown in the illustration for *dateFormat*. Do not specify units with *headerHeight*.
interval	Integer giving the number of minutes in each time slot on the calendar. The default *interval* is 30; the minimum is 5. The *interval* value can be a literal string, or it can contain a Zen #()# runtime expression.
monthList	Comma-separated list of the full names of months of the year. The default *monthList* value uses the $$$Text macro with the English values January through December. You can use $$$Text with *monthList* because its ZENLOCALIZE datatype parameter is set to 1 (true). Any localized *monthList* string must remain a comma-separated list of twelve items when translated.
OnGetScheduleInfo	Name of a server-side callback method in the Zen page class. This method gets the information needed to display the schedule. For details, see the section "<schedulePane> OnGetScheduleInfo Callback Method."

Attribute	Description

onselectitem	The *onselectitem* event handler for the <schedulePane>. Zen invokes this handler when the user clicks on an item within the schedule calendar. See "Zen Component Event Handlers."
	
	An *onselectitem* definition generally looks like this:
	
	`<schedulePane onselectitem="zenPage.selectItem(id,time);" ... >`
	
	While the signature for the *onselectitem* handler method in the Zen page class looks like this:
	
	Class Member
	
	`ClientMethod selectItem(id, time)[ Language = javascript ]`
	
	The handler uses the following two special variables as arguments:
	• `id` — the user defined identifier for a scheduled item as provided by the *OnGetScheduleInfo* callback (or null for an empty cell).
	• `time` — the time value associated with the cell (in yyyy-mm-dd hh:mm:ss format).

shortDayList	Comma-separated list of the abbreviated names of days of the week. The default *shortDayList* value uses the $$$Text macro with the following string: `Sun,Mon,Tue,Wed,Thu,Fri,Sat` You can use $$$Text with *shortDayList* because its ZENLOCALIZE datatype parameter is set to 1 (true). Any localized *shortDayList* string must remain a comma-separated list of seven items when translated. Be sure to coordinate *shortDayList* changes with *dayList* and *firstDayOfWeek*.
shortMonthList	Comma-separated list of the abbreviated names of months of the year. The default *shortMonthList* value uses the $$$Text macro with the following string: `Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec` You can use $$$Text with *shortMonthList* because its ZENLOCALIZE datatype parameter is set to 1 (true). Any localized *shortMonthList* string must remain a comma-separated list of twelve items when translated.
startTime	Starting time for the daily time slots to be displayed on the schedule calendar. This is an integer expressing the number of minutes since midnight. The default *startTime* is 480, which means 8 a.m. The range for *startTime* is 0 through 1440, which covers all 24 hours in the day. If the end user schedules an appointment that falls outside the *startTime* and *endTime*, the calendar expands to display that time slot. The *startTime* value can be a literal string, or it can contain a Zen #()# runtime expression.
view	Specifies what type of schedule to display. Possible values are `"day"`, `"week"`, or `"month"`. The default is `"day"`. The *view* value can be a literal string, or it can contain a Zen #()# runtime expression.

When you work with %ZEN.ComponentEx.schedulePane programmatically, you must also know about the following properties of the schedulePane class:

- Each <parameter> element provided in the original <schedulePane> definition in XData Contents becomes a member of the schedulePane parameters property, a list collection of %ZEN.Auxiliary.parameter objects. Each <parameter> acquires an ordinal position in the parameters collection: 1, 2, 3, etc.
- The read-only selectedTime property holds the start time of the currently selected time slot in the schedule calendar. The selectedTime is an integer that gives the number of seconds since 0, at midnight.
- The read-only selectedInterval property holds the length, in minutes, of the currently selected time slot in the schedule calendar. Add the selectedInterval to the selectedTime to calculate an end time.

9.10 Finder Pane

The <finderPane> component implements a simple browser for hierarchically organized data. Data must be supplied in JavaScript Object Notation (JSON). The Zen component <altJSONProvider> provides data in this format. See the section "Zen JSON Components" in the book *Developing Zen Applications*. (But also see *Using JSON in Caché*.)

The sample class ZENTest.FinderPaneTest in the SAMPLES namespace, illustrates the use of <finderPane> with <altJSONProvider>. The following example shows the code that creates the JSON component, and sets the *OnGetArray* callback to the method `GetFinderArray`, which is defined in the class.

XML

```
<altJSONProvider id="json" OnGetArray="GetFinderArray"/>
```

The next example shows the code that creates the finder pane. It sets the *ongetdata* property of <finderPane> to the *getContentObject* method of the <altJSONProvider> component. This method returns the client-side version of the data supplied by the <altJSONProvider>. The properties *onselectitem*, *ondrawdetails*, and *ondrawempty* are set to methods defined in the class.

XML

```
<finderPane id="finder"
        ongetdata="return zen('json').getContentObject();"
        onselectitem="return zenPage.itemSelected(item);"
        ondrawdetails="return zenPage.drawDetails(item);"
        ondrawempty="return zenPage.drawEmptyFinder();" />
```

<finderPane> has the following attributes:

Attribute	Description
Zen component attributes	<finderPane> has the same general-purpose attributes as any Zen component. For descriptions, see these sections: • "Behavior" in the "Zen Component Concepts" chapter of *Using Zen* • "Component Style Attributes" in the "Zen Style" chapter of *Using Zen*
animate	If true, then animate the appearance of the finder. The default is true.

Attribute	Description
caption	Text to display as a caption along the top of the finder. The text is *not* HTML escaped, so it can contain markup. The *caption* value can be a literal string, or it can contain a Zen #()# runtime expression. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. See "Zen Attribute Data Types."
columnWidth	Width of columns in the finder when in "columns" mode. The default is 200 pixels. Use the *viewType* attribute to select "columns" mode.
folderIcon	The default icon to display for folder items in "icons" mode. Use the *viewType* attribute to select "icons" mode. The default value is the filename of an image file suppled by the system. You can use this attribute to set a different default image. The filename specifies the path from the csp/broker directory under the Caché install directory. The *ongeticon* event handler can return different image files for items displayed in the finder.
hilightTop	If true, then apply a high light color to the top-level rows in "list" mode. The default is false.
itemIcon	The default icon to display for items in "icons" mode. Use the *viewType* attribute to select "icons" mode. The default value is the filename of an image file suppled by the system. You can use this attribute to set a different default image. The filename specifies the path from the csp/broker directory under the Caché install directory. The *ongeticon* event handler can return different image files for items displayed in the finder.
listColumns	If defined, this is a list of properties that supply the column values in 'list' mode.
oncancel	The *oncancel* event handler: If defined, handles the event fired when the user presses the escape key within the finder.
ondblclick	The *ondblclick* event handler: If defined, handles the event fired when the user double-clicks on an item within the finder.
ondrawdetails	The *ondrawdetails* event handler: If defined, handles the event fired when an item with no children is selected. If this event handler returns a value, then it is used as DHTML to render the item details.
ondrawempty	The *ondrawempty* event handler: If defined, handles the event fired when there is no data available to display within the finder. If this event handler returns a value, then it is used as DHTML providing content for the empty finder.
ondrawitem	The *ondrawitem* event handler: If defined, handles the event fired when an item within the finder is about to be drawn. If this event handler returns a value, then it is used as DHTML to render the item contents.
ongetdata	The *ongetdata* event handler: This defines the client-side code that returns a graph of javascript objects used to provide the contents of the finder.

Attribute	Description
ongeticon	The *ongeticon* event handler. If defined, handles events fired when the finder is in "icons" view and returns the url of the icon to use for the current item. If it returns "" (the empty string), then the default icon is used. The current item is passed to the event handler as an item.
onlazyload	The *onlazyload* event handler: Used to partially load data into the finder. This defines the client-side code that returns a graph of javascript objects that are used as the children of the current node.
onselectitem	The *onselectitem* event handler. If defined, handles the event fired when the user clicks on an item within the finder.
parameters	User-defined set of parameters. These are currently not used by the finder.
selectedList	This is a list of numbers (0-based) indicating the current selected item(s). The first number is the index in the top-most list of items; the second is the index within the children of the top-most item and so on.
upIcon	The default icon to display for the button that enables users to move up a level when the finder is in “icons” mode. Use the *viewType* attribute to select “icons” mode. The default value is the filename of an image file suppled by the system. You can use this attribute to set a different default image. The filename specifies the path from the csp/broker directory under the Caché install directory.
viewType	How the contents of the finder component are displayed. Possible values are: • icons — arranges items in a grid, with an image representing each item. • list — arranges items in a vertical list. • columns — arranges items in columns. The default value is “columns”.