Developing Zen Applications

Version 2018.1
2024-05-02

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

Developing Zen Applications
Caché Version 2018.1 2024-05-02

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

List of Figures

List of Tables

About This Book

This book addresses advanced Zen application programming issues such as security, localization, client side layout management, and custom components.

This book contains the following sections:

- "Zen Applications" provides details about Zen applications.
- "Zen Pages" provides details about Zen page classes.
- "Zen Security" explains how to provide security at the application, page, or component level.
- "Zen Localization" explains how to substitute translated messages for different language locales.
- "Custom Components" explains how to fit new components into the Zen framework.
- "Client Side Layout Managers" explains how users can adjust page layout at runtime.
- "Client Side Menu Components" describes menu components that are easily fine-tuned.
- "Client Side Library" describes the Zen library of client side functions and properties.

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 Zen Components* details the simple built-in Zen components for web application development.
- *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 a client and a server.
- *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*.

1

Zen Applications

A Zen application specifies the full set of activities that can result when a Zen client and server interact. These activities may consist of displaying web pages, issuing queries to a database, writing data to disk, and more. When you create a Zen application, you create a suite of Zen classes that specify these activities. The language you use for these classes is ObjectScript — with embedded snippets of XML, HTML, SVG, and JavaScript as needed.

This chapter explores Zen application programming based on the foundation provided by *Using Zen*. It provides the following topics:

- Zen Classes as CSP Classes
- Zen Application Configuration
- Zen Application Classes
- Sample Development Project

1.1 Zen Classes as CSP Classes

The base application class %ZEN.application and the base page class %ZEN.Component.page each inherit from %CSP.page. This means that every Zen application class and every Zen page class is also an instantiable Caché Server Page. All the CSP page class properties, methods, parameters, and variables (such as `%session`) are available to Zen application and page classes.

Important: Experienced CSP programmers should note there are important differences between Zen and CSP classes. For example, the allowed expressions that can appear within #()# syntax is much more restrictive in Zen than it is in CSP. For details, see the section "Zen Runtime Expressions."

1.2 Zen Application Configuration

A Zen application may require configuration of its Caché system settings. In order to accomplish this, you need to configure the web application associated with the Zen application. This section explains what this means, and how to perform the configuration steps.

Zen application code resides in a Caché namespace. Each Caché namespace has at least one web application mapped to it. By default, when you create a new namespace called *myNamespace* it comes with a web application already mapped to it. This web application has default settings already in place, including the following settings which are significant for Zen:

- **Web application Name** — The application name. The default value is:

  `/csp/`*myNamespace*

  A web application of this name is automatically created each time you create a new Caché namespace for any purpose.

- **CSP Files Physical Path** — The location on the server where the application stores its files. The default value is:

  install-dir`/CSP/`*myNamespace*

  Where *install-dir* is the Caché installation directory.

  At compile time, a Zen class generates JavaScript (.js) and Cascading Style Sheet (.css) files. It may also include external .js or .css files at compile time. If the Zen class is a custom component, it uses the INCLUDEFILES class parameter to list external files, and if it is an application or page, it uses the CSSINCLUDES or JSINCLUDES class parameters to do so. To specify the files, you can use full URLs or simple file names. In the latter case, Zen assumes that the path to these files is the **CSP Files Physical Path** for the default web application for the Caché namespace.

  Usually a Zen application uses all default settings for its Caché namespace and associated web application. This means that usually you do not need to configure the **CSP Files Physical Path**. Everything works without your intervention. However, if you want to find the generated files, or supply external .js or .css files to use with your Zen application, then you need to know what the path is: It is the **CSP Files Physical Path**.

- **Default Timeout** — Number of seconds of inactivity that cause the Zen application to time out.

- **Serve Files** — For Zen applications, InterSystems strongly recommends that you set this value to "Always" or "Always and cached." Do not set it to "No."

  If you do not wish to serve files from Caché, you must ensure there are copies of all the Zen static files in each directory that is using Zen. The Zen static files are found in the `/csp/broker` directory under the Caché installation directory. Place a copy of each of these files on your web server in the directory associated with each Zen application. If you are serving files from Caché, you do not need to do this.

- **Custom Error Page** — The page that is displayed when the Zen application encounters an error that it cannot resolve, such as a page missing from the application. By default, this field is blank and the default error page for the browser displays. When a value is supplied for this field, it must be a URL in the appropriate format; that is, it must be the **Web application Name** followed by the package and class name of the Zen page to display, for example:

  `/csp/`*myNamespace*`/MyDemo.Error.csp`

- **Login Page** — The URL of the page to use as the application login page. This URL must begin with the **Web application Name** followed by the package and class name of the Zen page to display, for example:

  `/csp/`*myNamespace*`/MyDemo.Login.csp`

  The **Change Password Page** is similar to **Login Page**, but specifies the page that allows a user to change his or her password. For a discussion and examples, see "Controlling Access to Applications" in the chapter "Zen Security."

At compile time, a Zen application finds its associated web application and applies its configuration settings. That is, Zen cascades through the following sequence and returns the first web application that it finds:

1. The Zen application knows which Caché namespace it resides in, for example *myNamespace*.
2. If there is a web application in this namespace whose name is `/csp/`*myNamespace*, Zen uses the settings from this web application.

3. If not, Zen searches for any web application mapped to *myNamespace* and returns one based on standard Caché global collation order. For example, if there are web applications /A and /X that map to *myNamespace*, Zen uses the settings from application /A.

Important: There is no way to configure an explicit association between a Zen application and a web application. The simplest way to manage the implicit association is to keep each Zen application in its own Caché namespace that has no other application in it and, when you need to configure application settings, configure the default web application for that namespace.

Whether or not you actually need to configure web application settings depends on the features that you add to your Zen application. The beginning of this section lists the settings you most often need to adjust. When you need to configure web application settings, the basic procedure is as follows:

- Start the Management Portal.
- Navigate to the **Web Applications** page (**System Administration** > **Security** > **Applications** > **Web Applications**).
- Find the appropriate web application in the list and click its **Edit** button. The **Edit Web Application** page displays.
- Make changes as needed, and click **Save**.

For High Availability solutions running over CSP, InterSystems recommends that you use a hardware load balancer for load balancing and failover. InterSystems requires that you enable sticky session support in the load balancer. This guarantees that — once a session has been established between a given instance of the gateway and a given application server — all subsequent requests from that user run on the same pair.

This configuration assures that the session ID and server-side session context are always in sync; otherwise, it is possible that a session is created on one server but the next request from that user runs on a different system where the session is not present, which results in runtime errors (especially with hyperevents, which require the session key to decrypt the request). To enable sticky session support, see your load balancer documentation.

Note: It is possible to configure a system to work without sticky sessions, but this requires that the CSP session global be mapped across all systems in the enterprise and can result in significant lock contention so it is not recommended.

1.3 Zen Application Classes

A Zen application class is a class derived from %ZEN.application that provides high-level housekeeping and acts as the single root for all of the pages in the application. The application class does not keep an inventory of its associated pages. The association between the application and its pages occurs because every Zen page class has an optional parameter called APPLICATION that identifies the application class. This parameter is optional because an application class is optional. Your Zen application needs to provide an application class only if you consider it a useful convenience. However, if you do provide an application class, you may provide *only one* application class and every page in your application must identify this application using the APPLICATION parameter.

The following table lists the elements of a Zen application class.

Table 1–1: Application Class Elements

Element	Role	Specific Items	Description
Class parameters	General application settings	APPLICATIONNAME	Defines a text string that you can use in titles or labels.
		CSSINCLUDES	Comma-separated list of Cascading Style Sheet (.css) files to include for every page in the application. You can use URLs or simple file names. If you use simple file names, the **CSP Files Physical Path** specifies the physical path to these files. For further information, see the section "Zen Application Configuration." For information about style sheets, see the "Zen Style" chapter in the book *Using Zen*.
		HOMEPAGE	URI of a default Home Page for the application. This makes it possible for the application class to be specified as a starting point for the application, even though it does not itself display a page. When the application class receives an HTTP request, it redirects the request to the URI specified by the HOMEPAGE class parameter.
		JSINCLUDES	Comma-separated list of JavaScript (.js) include files to include for every page in the application. You can use URLs or simple file names. If you use simple file names, the **CSP Files Physical Path** specifies the physical path to these files. For further information, see the section "Zen Application Configuration."
		USERPACKAGES	Comma-separated list of user-defined class packages whose HTML class and style definitions are in pre-generated include files. These include files are available to every page within the application.
		USERSVGPACKAGES	Comma-separated list of user-defined class packages whose SVG class and style definitions are in pre-generated include files. These include files are available to every page within the application.
An embedded XML document	CSS style definitions	XData Style	Any CSS style definitions that appear within this XData block are placed on every page within the application. See the "Zen Style" chapter in the book *Using Zen*.

When programming, be aware that the application class can contain server-side methods written in ObjectScript, Caché Basic, or Caché MVBasic only. It cannot contain any client-side methods marked with the javascript keyword, or client/server methods marked with the ZenMethod keywords. The application class can execute methods on the server only.

1.4 Sample Development Project

The chapter "Zen Layout" in the book *Using Zen* explains how to use template pages and panes to organize layout and style for a Zen application. This topic traces an example that uses composites and dynamic page generation to organize behavior for a Zen application. This example uses Zen to add new order entry module pages to an existing Weblink application. Constraints are as follows:

- You need to be able to call the new pages from within your existing application.
- You do not have the luxury of rewriting the whole application, so you want to evolve into Zen one module at a time.
- Your order entry page needs to have flexible content. That is, you need to be able to easily update portions of the page depending on the context.
- Your order entry page is so big and complex, you want to only build what you need when the page is first loaded and then fill in some empty segments on the fly.
- Ideally, you want to break up the page into segments that can be written simultaneously by different developers.

The following example addresses these constraints. It purposely does not address layout or style but focuses on behavior.

- The following class provides the main page. It defines a set of named groups. Each group is populated with components from the server in response to user events. The main page also defines an API (set of methods) that the various components can call to work with the page. For example, there is a method that loads a new component into one of the groups on the page. Note the parameter USERPACKAGES, the groups `g1` and `g2`, and the method **loadForm**, which adds a specific type of form to the page based on the parameters provided to it.
- The contents of each group is defined using a composite component. A composite is a custom component that you build by grouping one or more built-in Zen components in a specific way. For full details, see the "Composite Components" section in the chapter "Custom Components." Composites can be developed individually by different developers; this satisfies the project constraints. In the sample code, there are four such composites: searchForm is the starting form that drives the loading of other forms. formOne, formTwo, and formThree are sample forms.

Class Definition

```
Class TestMe.MainPage Extends %ZEN.Component.page
{
  /// Comma-separated list of User class packages whose HTML class
  /// and style definitions are in pre-generated include files.
  Parameter USERPACKAGES = "TestMe.Components";

  /// Class name of application this page belongs to.
  Parameter APPLICATION;

  /// Displayed name of this page.
  Parameter PAGENAME;

  /// Domain used for localization.
  Parameter DOMAIN;

  /// This Style block contains page-specific CSS style definitions.
  XData Style
  {
    <style type="text/css">
      /* style for title bar */
      #title {
        background: #C5D6D6;
        color: black;
        font-family: Verdana;
```

```
        font-size: 1.5em;
        font-weight: bold;
        padding: 5px;
        border-bottom: 1px solid black;
        text-align: center;
      }

      /* container style for group g1 */
      #g1 {
        border: 1px solid black;
        background: #DDEEFF;
      }

      /* container style for group g2 */
      #g2 {
        border: 1px solid black;
        background: #FFEEDD;
      }
    </style>
  }

  /// This XML block defines the contents of this page.
  XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
  {
    <page xmlns="http://www.intersystems.com/zen"
          xmlns:testme="TestMe"
          title="">
      <html id="title">TestMe Test Page</html>
      <vgroup id="gSearch" width="100%">
        <testme:searchForm/>
      </vgroup>
      <spacer height="40"/>
      <!-- these groups are containers for dynamically loaded components -->
      <hgroup>
        <vgroup id="g1"/>
        <spacer width="20"/>
        <vgroup id="g2"/>
      </hgroup>
    </page>
  }

  /// Create a form (via a composite element) and place it into the group with
  /// the id groupId. formClass is the name of the composite element
  /// containing the form. formId is the id applied to the form.
  ClientMethod loadForm(formClass, formId, groupId) [ Language = javascript ]
  {
    try {
      // if there is already a form, get rid of it
      var comp = zen(formId);
      if (comp) {
        zenPage.deleteComponent(comp);
      }

      var group = zen(groupId);
      var form = zenPage.createComponentNS('TestMe',formClass);
      form.setProperty('id',formId);
      group.addChild(form);
      group.refreshContents(true);
    }
    catch(ex) {
      zenExceptionHandler(ex,arguments);
    }
  }
}
```

Some important things to notice about the TestMe.MainPage class:

- The parameter USERPACKAGES is set as follows:

  ```
  Parameter USERPACKAGES = "TestMe.Components";
  ```

  This tells the page to use the generated include files for the components in this package. This is important because we are creating objects on the fly, and we want to ensure that the relevant JavaScript objects are available.

- The XData Contents block for the page provides a reference to the XML namespace for the new composite components:

  ```
  xmlns:testme="TestMe"
  ```

- XData Contents places the searchForm component on the page as follows:

  ```
  <testme:searchForm/>
  ```

When the user presses one of the buttons on the searchForm, it invokes the loadForm method on the Main page. This deletes the current form (by id) and loads a new form (it does not actually have to be a form, that is just what we have used as an example).

Now take a look at the components for the example. As suggested in "Composite Components," all of these composite component classes are placed together in their own package, TestMe.Components. The reason for this is that Zen automatically generates JS and CSS header files for components in a given package. By placing these components in a package you can exploit this feature by setting the parameter USERPACKAGES to the package name, on the main page, as shown above:

```
Parameter USERPACKAGES = "TestMe.Components";
```

The composites also use their own XML namespace to avoid conflict with other components.

The composite component classes are as follows:

- searchForm — the starting form that drives the loading of other forms:

Class Definition

```
Class TestMe.Components.searchForm Extends %ZEN.Component.composite
{
  /// This is the XML namespace for this component.
  Parameter NAMESPACE = "TestMe";

  /// This Style block contains component-specific CSS style definitions.
  XData Style
  {
    <style type="text/css">
    </style>
  }

  /// Contents of this composite component.
  XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
  {
    <composite labelPosition="left">
      <button caption="Show FormOne in Group 1"
              onclick="zenThis.composite.btnForm1();"/>
      <button caption="Show FormTwo in Group 1"
              onclick="zenThis.composite.btnForm2();"/>
      <button caption="Show FormTwo in Group 2"
              onclick="zenThis.composite.btnForm3();"/>
      <button caption="Show FormThree in Group 2"
              onclick="zenThis.composite.btnForm4();"/>
    </composite>
  }

  /// User click on form 1 button.
  ClientMethod btnForm1() [ Language = javascript ]
  {
    // Notify the page that the search button was pressed
    // and that a new form should be loaded.
    zenPage.loadForm('formOne','form1','g1');
  }

  /// User click on form 2 button.
  ClientMethod btnForm2() [ Language = javascript ]
  {
    // replace contents of 'g1' with formTwo
    zenPage.loadForm('formTwo','form1','g1');
  }

  /// User click on form 3 button.
  ClientMethod btnForm3() [ Language = javascript ]
  {
    // replace contents of 'g2' with formTwo
    zenPage.loadForm('formTwo','form2','g2');
  }

  /// User click on form 4 button.
  ClientMethod btnForm4() [ Language = javascript ]
  {
    // replace contents of 'g2' with formThree
    zenPage.loadForm('formThree','form2','g2');
  }
}
```

Every component that is part of a composite can refer to its containing composite group via its composite property. You can see this in the syntax used to invoke client-side methods defined in the composite class, when the user clicks a button on this form:

```
onclick="zenThis.composite.btnForm1();"
```

- formOne — a sample form:

Class Definition

```
Class TestMe.Components.formOne Extends %ZEN.Component.composite
{
  /// This is the XML namespace for this component.
  Parameter NAMESPACE = "TestMe";

  /// This Style block contains component-specific CSS style definitions.
  XData Style
  {
    <style type="text/css">
    </style>
  }

  /// Contents of this composite component.
  XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
  {
    <composite labelPosition="left">
      <titleBox title="Form One"/>
      <text id="ctrlF1" label="Search:"/>
    </composite>
  }
}
```

- formTwo — another sample form:

Class Definition

```
Class TestMe.Components.formTwo Extends %ZEN.Component.composite
{
  /// This is the XML namespace for this component.
  Parameter NAMESPACE = "TestMe";

  /// This Style block contains component-specific CSS style definitions.
  XData Style
  {
    <style type="text/css">
    </style>
  }

  /// Contents of this composite component.
  XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
  {
    <composite labelPosition="left">
      <titleBox title="Form Two"/>
      <text id="ctrlF1" label="F1:"/>
      <text id="ctrlF2" label="F2:"/>
      <text id="ctrlF3" label="F3:"/>
      <text id="ctrlF4" label="F4:"/>
    </composite>
  }
}
```

- formThree — yet another sample form:

Class Definition

```
Class TestMe.Components.formThree Extends %ZEN.Component.composite
{
  /// This is the XML namespace for this component.
  Parameter NAMESPACE = "TestMe";

  /// This Style block contains component-specific CSS style definitions.
  XData Style
  {
    <style type="text/css">
    </style>
  }
```

```
  /// Contents of this composite component.
  XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
  {
    <composite labelPosition="left">
      <titleBox title="Form Three"/>
      <colorPicker
 onchange="zenThis.composite.colorChange(zenThis.getValue());"/>
    </composite>
  }

  /// colorChange
  ClientMethod colorChange(color) [ Language = javascript ]
  {
    alert('You have selected: ' + color);
  }
}
```

2

Zen Pages

Pages within a Zen application are derived from the class %ZEN.Component.page. A page class defines and serves the contents of a Zen page in a browser. This chapter explores Zen page programming. It begins from the foundation provided by:

- The book *Using Zen*
- The previous chapter in this book: "Zen Applications"

This chapter provides technical details to support the information listed above. Topics in this chapter include:

- Zen Page Contents
- Zen Methods on Client and Server
- Server Side Methods
- Client Side Methods
- Zen Properties on Client and Server
- Zen Page Class Parameters
- Zen Special Variables
- Client Side Functions, Variables, and Objects
- Zen Runtime Expressions
- Zen Proxy Objects
- Zen JSON Components
- Zen Page Event Handling
- Zen Page URI Parameters
- Zen Layout Handlers
- Zen Utility Methods

Important: A complex Zen page requires a long serial state header string to be sent with each request. Note that there is a fixed 32k limit to the length of this header string, resulting in a message like the following:

```
"An error has occured on the server: ERROR #5001: Serial State Header is too large"
```

ZenMethods that are class methods do not require a serial state header, so you may be able to avoid the limit by using class methods. However, if a Zen method needs to use the *%page* variable, that method must be an instance method, because *%page* is not available in class methods.

2.1 Zen Page Contents

The visible contents of a page are defined by a set of Zen components that belong to the page. This is the *component object model* for the page. To review background information and diagrams that illustrate the component object model and its relationship to the page, see the "Zen Application Concepts" chapter in the book *Using Zen*.

The sections in this topic explain how to specify the component object model for a Zen page. The following list of techniques contains links to the sections that provide details. You can specify Zen page contents as follows:

- Defining Page Contents Statically Using XML You can provide an XML block named XData Contents in the Zen page class.
- Modifying Page Contents Prior to Display You can override the **%OnAfterCreatePage**() callback method so that it invokes server-side methods that retrieve pointers to components, add components, modify components, hide or redisplay components, and remove components from the component object model.
- Modifying Page Contents after Display. After the page has displayed, set up handlers for user actions (such as clicks) so that they invoke client-side or server-side methods that retrieve pointers to components, add components, modify components, hide or redisplay components, or remove components from the component object model.

You can also override the **onloadHandler**() client-side method to modify the page before it displays on the client side. This JavaScript method runs on the client side just prior to displaying the page. See the entry for **onloadHandler**() in the table in section "Client Side Callback Methods for the Page" for more information.

You can define the basic page using XML, then use any combination of the above techniques to modify it dynamically at runtime.

2.1.1 Defining Page Contents Statically Using XML

A Zen page class can provide an `XData Contents` block (an XML document embedded within the class definition) that defines the set of components used for the page. It is possible to define the contents of a page programmatically, but in most cases an XML block is more convenient.

The following `XData Contents` block defines a simple page object containing a single button component:

Class Member

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
<page xmlns="http://www.intersystems.com/zen" title="My Page">
  <button caption="Hey"/>
</page>
}
```

The `XData Contents` block is not processed at runtime. Instead, at compile time the class compiler uses the `XData Contents` information to generate a method called **%CreatePage**. You can view (but not modify) this method if you like

by compiling a page class and then using the View Other command in Studio to see the code generated for the page class. The **%CreatePage** method is called at runtime to create the contents of the page component model.

For a detailed description of XData Contents, see the chapters "Zen Layout" and "Zen Style" in the book *Using Zen*.

Note: InterSystems recommends that you include an XML namespace declaration within an XData Contents definition. You do this by providing the *xmlns* attribute with the <page> element. For example:

```
<page xmlns="http://www.intersystems.com/zen" >
```

As you add custom components, chances increase for conflicts between components in different namespaces. This is also true for other Zen elements such as <composite> and <pane>. Adding the *xmlns* attribute to the <page> prevents namespace conflicts.

2.1.2 Modifying Page Contents Prior to Display

If you need to programmatically modify the contents of a page before displaying it, add code to the **%OnAfterCreatePage**() callback method as in the following example. This example creates a new group component, then adds the group to the page using the **%AddChild**() method of the page. Subsequently, it creates several components, sets their properties, and adds them to the group using **%AddChild**() method of the group:

Class Member

```
Method %OnAfterCreatePage() As %Status
{
  Set tGroup = ##class(%ZEN.Component.group).%New()
  Do %page.%AddChild(tGroup)

  Set tBtn = ##class(%ZEN.Component.button).%New()
  Set tBtn.caption = "Button 1"
  Do tGroup.%AddChild(tBtn)

  Set tBtn = ##class(%ZEN.Component.button).%New()
  Set tBtn.caption = "Button 2"
  Do tGroup.%AddChild(tBtn)

  Quit $$$OK
}
```

Any text you add for captions or other labels is *not* automatically localized as it is when you add a caption attribute using XML. If you want localized captions, your **%OnAfterCreatePage**() method must call the $$$Text localization macros. This is not shown in the example above.

If the component is not visible (such as <timer>) you may add it to a page using **%AddComponent**() instead of **%AddChild**(). For components that are visible, the methods **%AddChildBefore**() and **%AddChildAfter**() are also available. These methods have a second parameter that identifies the existing, sibling component.

One of the best ways to get a sense of how pages are put together programmatically is to look at the generated code for the **%CreatePage** method in any Zen page class that uses XData Contents to define page contents. To do this:

1. Open the page class in Studio.
2. Choose **View > View Other** or **Ctrl-Shift-V** or the  icon.
3. Select the file marked "1" and click **Open**.
4. Search for this string: `%CreatePage`
5. Scroll through the method to see how components are added to the page.

The following table lists the server-side methods that allow you to dynamically manipulate the component object model for a page. These methods are final and you cannot override them. All group components, including the page, support these methods. You can invoke these methods on a group or page object at any hierarchical level in the component object model.

Table 2–1: Page and Group Server Side Methods for Dynamic Page Contents

Method	Description
%AddChild(*obj*)	Add a child component to this page or group. *obj* is a pointer to the component you wish to add. If **%AddChild**() is called on a group, it adds the component to the group and also adds it to page that the group belongs to. Before calling **%AddChild**() to add a component to a group: • The group itself must be added to the page. • The component object must exist; create it using **%New**() • The id property of the component must be set.
%AddChildAfter(*obj*,*sib*)	Same functionality and restrictions as **%AddChild**(), but adds the child component *obj* to the group in the position immediately after the sibling object identified by *sib*. *sib* must be a pointer to an object that is already in the group.
%AddChildBefore(*obj*,*sib*)	Same functionality and restrictions as **%AddChild**(), but adds the child component *obj* to the group in the position immediately before the sibling object identified by *sib*. *sib* must be a pointer to an object that is already in the group.
%AddComponent(*obj*)	*obj* is a pointer to the component you wish to add to the page or group. Use **%AddComponent**() as an alternative to **%AddChild**() only when adding an invisible component, such as a <timer>. Otherwise use **%AddChild**().
%GetChildIndex(*obj*)	Look for the child component *obj* within this group and return its 1-based index number. Return -1 if unable to find the child.
%RemoveChild(*obj*)	Remove child component *obj* from this group. Returns 1 (true) if the component was found and removed.
%RemoveChildren(*flag*)	Remove all child components from this group. If *flag* is 1 (true) then remove only those children that were dynamically added to the group, retaining those that were statically added from XData Contents. If *flag* is omitted or 0 (false) delete all child components from this group.

2.1.3 Modifying Page Contents after Display

If you need to modify components in a page dynamically based on user interactions on the client side, you can do this as well. The SAMPLES namespace class ZENTest.DynamicComponentsTest provides useful examples of doing this from both the client and the server sides in response to user input. The "Sample Development Project" section provides another example of dynamically adding components to a page. Remember that you need to both create the component *and* add it as a child of the page before the user can see it.

In order for components to display correctly on a Zen page, Zen must load the relevant CSS and JavaScript. In the interest of improving performance, Zen pages do not automatically load CSS and JavaScript for all components. Zen determines which files to load by parsing the XData Content block and methods called during display of the page, such as **%OnAfterCreatePage**(). When you add components to the page dynamically in a client- or server-side method, you must use the %import property to identify the components so Zen can load the required CSS and JavaScript files. Zen checks for redundancy, so if the import property specifies a file that has already been loaded, the file is not included twice.

The following code illustrates an %import property for the <svgFrame> and <lineChart> components.

```
Property %import As %ZEN.Datatype.string(MAXLEN = 1000, XMLNAME = "import")
  [InitialExpression = "%ZEN.SVGComponent.svgFrame,%ZEN.SVGComponent.lineChart"];
```

If a page displays correctly even though you have not used %import for all dynamically added components, you should not assume that using %import is not necessary. The required CSS and JavaScript may have been included in files loaded as a result of parsing the XData Content block. However, InterSystems may change the organization of these CSS and JavaScript files at any time.

The following tables list the client-side methods that allow you to dynamically manipulate the component object model for a page. These methods are final and you cannot override them. The page supports the following client-side methods for creating and deleting component objects.

Table 2–2: Page Client Side Methods for Creating Components

Method	Description
createComponent(*name*)	Create a new component object in the default zen namespace. *name* is the simple name of a component class, for example "button".
createComponentNS(*ns*,*name*)	Create a new component object in the *ns* namespace. *name* is the simple name of a Zen component class — for example "button" — or the full package and class name of a custom component. When you use this method, it is important to also use one of the following conventions to ensure that Zen knows how to load the JavaScript for the custom components: • The *import* component attribute, as described in the section "Component Style Attributes" in the "Zen Style" chapter of Using Zen. • The USERPACKAGES class parameter, as shown in the "Sample Development Project" section, and as described in Zen Application Classes and "Zen Page Class Parameters."
deleteComponent(*obj*,*rfr*,*sync*)	Given the component identified by *obj*, delete it from the page. If the optional *rfr* (refresh) parameter is false, the page is not refreshed after the delete. This is useful if you know that subsequent code causes such a refresh. The default is to refresh. If the optional *sync* (synchronize) parameter is true, the refresh of the group containing the deleted component is executed synchronously. The default is asynchronous.

All group components, including the page, support client-side methods for adding and removing component objects once they are created. You can invoke these methods on a group component at any hierarchical level in the component object model, including the page itself.

Table 2–3: Page and Group Client Side Methods for Dynamic Page Contents

Method	Description
addChild(*obj*,*refresh*)	Add a child component to this page or group. *obj* is a pointer to the component you wish to add. You must first create the component using **createComponent**() or **createComponentNS**(). If *refresh* is true, refresh the display of this group after adding the new component.
addChildAfter(*obj*,*sib*,*refresh*)	Same as **addChild**(), but adds the child component *obj* to the group in the position immediately after the sibling object identified by *sib*. *sib* must be a pointer to an object that is already in the group. If *refresh* is true, refresh the display of this group after adding the new component.

Method	Description
addChildBefore(*obj*,*sib*,*refresh*)	Same as **addChild**(), but adds the child component *obj* to the group in the position immediately before the sibling object identified by *sib*. *sib* must be a pointer to an object that is already in the group. If *refresh* is true, refresh the display of this group after adding the new component
getChildIndex(*obj*)	Look for the child component *obj* within this group and return its 01-based index number. Return -1 if unable to find the child.
removeChild(*obj*)	Remove child component *obj* from this group. Returns 1 (true) if the component was found and removed.

2.2 Zen Methods on Client and Server

The following table outlines the possibilities for different types of methods in a Zen page class.

Table 2–4: Page Class Method Conventions

Callable From	Runs On	Class Scope	Timing	Code Language	Keywords	Naming Convention
Client	Client	Instance	—	JavaScript	ClientMethod [Language = javascript]	myMethod
Client	Server (Can send back code that runs on the client)	Instance	Async	ObjectScript, Caché Basic, Caché MVBasic	[ZenMethod]	MyMethod
			Sync			
		Class	Async			
			Sync			
Server	Server	Instance	—	ObjectScript, Caché Basic, Caché MVBasic	—	%MyMethod
		Class				

Any method defined within a page class whose name does *not* start with % becomes a method of the client-side page object. Any method whose name starts with % is server-only.

Any instance method defined with the keyword ClientMethod with `[Language = javascript]` becomes a client-side instance method of the client page object. When called, it executes within the browser.

Any method defined with its `[ZenMethod]` keyword set automatically becomes a client-side instance method of the client page object. When the client-side method is called, the server-side version of the method is executed. Any context needed to run the server method is automatically provided. An instance method runs as an instance method on the server; the serialized state of its client object is shipped to the server and instantiated. A class method runs as a class method on the server (but is called as an instance method on the client).

2.2.1 Synchronous and Asynchronous Methods

Server-side methods can be called synchronously or asynchronously. This happens very smoothly and automatically as follows: If a server-side method has a return type, it is called synchronously, and its return value is returned to the client. If the method has no return type, it is called asynchronously; the client does not wait for the return value.

The CSP hyperevent logic handles all the details of this, so (unlike AJAX) you do not need to provide any additional application logic to enable asynchronous calls. In this way, Zen takes the next step past most AJAX frameworks.

The main advantages of asynchronous methods are:

- The browser can keep working while the server request is processed
- You can call a server method, modify the client (say by displaying a message), and the server logic can update this message when it completes

Some of the Zen components take advantage of asynchronous methods. For example, the <dataCombo> control uses an asynchronous method to execute a server-side query. This allows the <dataCombo> to display a `loading...` message while the query is running. This message is replaced when the response from the server arrives, so the user does not see the message if the query is quick.

Zen offers a client-side flag **zenSynchronousMode** that you can set to `true` to force all methods to be executed synchronously, regardless of their return values. When `false`, methods are synchronous if they have a return value, asynchronous otherwise. When you want to set this flag:

1. First, get and store the current value of **zenSynchronousMode**.
2. Then, set the flag as you wish.
3. When your processing is done, restore the previous value of **zenSynchronousMode**.

In addition to this convention, the client-side **refreshContents** method (available for every component) offers a single optional argument whose value can be `true` or `false`. This flag specifies whether the client should refresh the component synchronously (`true`) or asynchronously (`false`). If this argument is omitted, the default is `false`.

2.2.2 Collections as Method Arguments and Return Types

It is possible to pass a collection between the client and server using a Zen class method. This section explains the conventions for doing so. Both lists and arrays are supported. For an example using lists, see the section "Using a JavaScript Method" in the "Zen Charts" chapter of *Using Zen Components*.

Important: For the conventions described in this topic to work, the `[ZenMethod]` must be a class method, declared using the `ClassMethod` keyword as shown in the examples. Collections cannot be arguments or return types for instance methods. Also, only collections of literals are supported this way. If an application needs to pass something more complex, it must define a Zen object class and use that instead.

2.2.2.1 Lists

To pass a list collection as an argument to a server-side Zen class method, use %ListOfDataTypes as the data type, as shown here:

Class Member

```
ClassMethod MyMethod(list as %Library.ListOfDataTypes) [ZenMethod]
{
  Set x = list.GetAt(1)
}
```

From the client, you can call this method by passing it a JavaScript Array object:

```
 var list = new Array();
 list[list.length] = 22;
 zenPage.MyMethod(list);
```

To return a list from the server, use %ListOfDataTypes as the return value for the server-side Zen class method:

Class Member

```
ClassMethod MyMethod() as %Library.ListOfDataTypes [ZenMethod]
{
  Set ret = ##class(%Library.ListOfDataTypes).%New()
  Do ret.Insert("red")
  Quit ret
}
```

From the client, this method returns a JavaScript Array object, which is 0–indexed. For example:

```
var list = zenPage.MyMethod();
alert(list[0]); // should show 'red'
```

2.2.2.2 Arrays

To pass an array collection as an argument to a server-side Zen class method, use %ArrayOfDataTypes as the data type, as shown here:

Class Member

```
ClassMethod MyMethod(arg as %Library.ArrayOfDataTypes) [ZenMethod]
{
  Set x = arg.GetAt("item1")
}
```

From the client, you can call this method by passing it a JavaScript associative array (an Object in JavaScript). For example:

```
var arr = new Object();
arr['item1'] = 22;
zenPage.MyMethod(arr);
```

To return an array from the server, use %ArrayOfDataTypes as the return value for the server-side Zen class method:

Class Member

```
ClassMethod MyMethod() as %Library.ArrayOfDataTypes [ZenMethod]
{
  Set ret = ##class(%Library.ArrayOfDataTypes).%New()
  Do ret.SetAt("red","color")
  Quit ret
}
```

From the client, this method returns a JavaScript associative array (an Object in JavaScript). For example:

```
var arr = zenPage.MyMethod();
alert(arr['color']); // should show 'red'
```

2.2.2.3 Example: Passing Collections Between Client and Server

The following Zen page class example shows the code conventions required for the following behavior:

- Clicking the top button sends a collection object from client to server.

- Clicking the bottom button retrieves a collection object from the server and sends it to the client.

```
Class MyApp.Test Extends %ZEN.Component.page
{
  XData Contents [XMLNamespace = "http://www.intersystems.com/zen"]
  {
    <page xmlns="http://www.intersystems.com/zen" title= "">
      <button caption="Ship array to server"
              onclick="zenPage.testClientToServer();" />
      <button caption="Get array from server"
              onclick="zenPage.testServerToClient();" />
    </page>
  }

  ClientMethod testClientToServer() [ Language = javascript]
  {
    var list = new Array();
    list[list.length] = 22;
    alert(zenPage.MyClientToServer(list));
  }

  ClassMethod MyClientToServer(pList As %ListOfDataTypes) As %String [ZenMethod]
  {
    Set ^%x = pList.GetAt(1)
    Quit ^%x
  }

  ClientMethod testServerToClient() [ Language = javascript]
  {
    var list = zenPage.MyServerToClient();
    alert(list.join('\n'));
  }

  ClassMethod MyServerToClient() As %ListOfDataTypes [ZenMethod]
  {
    Set pList = ##class(%ListOfDataTypes).%New()
    Do pList. Insert("RED")
    Do pList. Insert("GREEN")
    Do pList. Insert("BLOOP")
    Quit pList
  }
}
```

2.2.3 Embedded HTML and JavaScript

The following sample **%DrawHTML** method uses a syntax that you see commonly throughout the Zen documentation and sample code: The keyword `&html` followed by HTML statements enclosed in angle brackets <>. This is ObjectScript syntax for *embedded HTML* statements:

Class Member

```
Method %DrawHTML()
{
    &html<<div class="MyComponent">Message</div>>
}
```

ObjectScript offers a similar syntax for *embedded JavaScript* statements in server-side methods marked with the `[ZenMethod]` keyword. Within these methods, the keyword `&js` can be followed by JavaScript statements enclosed in angle brackets <>. While executing the method on the server, Zen finds the snippet of JavaScript code and sends this snippet to the client for execution. Note that you must escape any angle brackets used inside the brackets that enclose the embedded JavaScript. For example:

```
Method BtnClickMe() [ZenMethod]
{
   &js<alert('Client Method');>
   Quit
}
```

In addition to `&html<>` and `&js<>`, there is also an `&sql<>` convention for SQL statements. Each of these ObjectScript language conventions offers a useful shortcut for enclosing snippets of code within Zen application and page classes. For syntax details, see the definition of the & symbol in the *Caché ObjectScript Reference*.

Important: The “&js< ... >” should ONLY be used when working in Synchronous mode and interacting with Class-Methods where there is no Document Object Model (DOM) synchronization happening.

In an instance method, if you are modifying elements in the DOM, this code is returned by the hyperevent, bundled in to a function and executed in the browser immediately on return; then the DOM then happens, overwriting any and all changes made by the function. If you are calling Asynchronously, this is also a risk that these functions may not execute in the order you expect them to.

2.2.4 Automatic Type Conversion for Server Side Methods

The mechanism for handling type conversions when invoking a server-side method works as follows:

- When calling a server-side method, any %ZEN.Datatype.boolean arguments are converted from client-side true/false to server-side 1/0.
- If a server method has a %ZEN.Datatype.boolean return type, its return value are converted to a client-side true/false value.
- If a server method has a %ZEN.Datatype.list return type, its return value are converted to a client-side array.
- If a server method has a numeric return type, its return value are converted to a client-side numeric value (unless it the method returns "").
- Methods that take object-valued arguments behave well if the argument is missing or null.

2.3 Server Side Methods

The topics in this section describe important page and component methods that run on the server side.

2.3.1 Server Side Callback Methods for the Page

Callback methods are user-written methods within a Caché class that are triggered, not by a method call, but by specific system events. To distinguish callback methods, Caché developers give them names of the following form:

%OnEvent

where the keyword **Event** identifies the Caché system event that triggers the callback. If an event supports a callback method, the method controlling the event automatically executes the callback method, if it exists within the class.

You must never execute callback methods by calling them explicitly. Instead, provide an implementation of the method within your class so that Caché can invoke the method automatically when the time is right. If you want your class to do nothing in response to an event, omit any implementation or override of the corresponding callback method.

When an HTML page derived from %ZEN.Component.page is drawn, it invokes a number of callback methods that are executed on the server before the page content is sent to the browser. The following table lists these methods. A page class can override these methods to control the behavior of the page.

Table 2–5: Page Class Server Side Callback Methods

Callback Method	Description
%OnBeforeCreatePage()	This class method is called just before the server-side page object is created.

Callback Method	Description
%OnCreatePage()	This instance method is called just after the server-side page object is created but before any of its child components are created.
%OnAfterCreatePage()	This instance method is called after the server-side page object and all of its predefined child components are created. A subclass can override this method to add, remove, or modify items within the page object model, or to provide values for controls.
%OnDrawHTMLHead()	This instance method is called when the HTML for the page is being rendered just before the end of the HTML HEAD section of the page. This provides a way to inject additional content into the HEAD section of a page should this ever be necessary.
%OnDrawHTMLBody()	This instance method is called when the HTML for the page is being rendered close to the start of the HTML BODY section of the page (before any child components are rendered). This provides a way to inject additional content into the BODY section of a page should this ever be necessary.
%OnMonitorBackgroundTask(...)	This class method is called when the client checks on the progress of a background task and the task is still running. For details, see the section "Running Background Tasks on the Server."
%OnFinishBackgroundTask(...)	This class method is called when the client checks on the progress of a background task and the task is complete. For details, see the section "Running Background Tasks on the Server."

2.3.2 Server Side Methods for the Page

The page class contains a number of server-side methods that can be useful when working with the component object model on the server, running background tasks on the server, or processing form submits. These methods are final and you cannot override them.

The following table lists the general purpose server-side methods. For a list of the server-side page class methods that allow you to modify the component object model, see the section "Defining Page Contents Dynamically on the Server."

Table 2–6: Page Class Server Side Methods

Method	Description
%GetComponent(*index*)	Return a pointer to the component object, given its system-assigned index number *index*.
%GetComponentById(*id*)	Return a pointer to the component object, given its user-assigned *id* (the *id* value assigned to it in XData Contents or other user code).
%GetComponentByName(*name*)	Return a pointer to the component object, given its user-assigned *name* (the *name* value assigned to it in XData Contents or other user code). If multiple components have the same name on this page, return the first one found.
%GetValueById(*id*)	Return the *value* of a control component, given its user-assigned *id*.
%GetValueByName(*name*)	Return the *value* of a control component, given its user-assigned *name*. If multiple components have the same name on this page, return the first one found.

Method	Description
%EndBackgroundMethod()	Clean up after a method that was invoked as a background task. For details, see the section "Running Background Tasks on the Server."
%RunBackgroundMethod(...)	Invoke a method as a background task. The method runs in the background but the Zen page does not wait for the method to return. For details, see the section "Running Background Tasks on the Server."
%SetBackgroundMethodStatus(...)	Can be called from within a method that is running in the background, to update its own status information. For details, see the section "Running Background Tasks on the Server."
%SetErrorById(*id*, *msg*)	Set the error message for a component, given the *id* assigned to this component in XData Contents or other user code. *msg* is the error message.
%SetErrorByName(*name*, *msg*)	Set the error message for a component, given its user-assigned *name*. *msg* is the error message.
%SetValueById(*id*)	Set the *value* of a control component, given its user-assigned *id*.
%SetValueByName(*name*)	Set the *value* of a control component, given its user-assigned *name*.
%SetValuesByName(*array*)	Set the values of a set of control components, given an *array* of values subscripted by the user-assigned *name* values of each control component in the set.

2.3.3 Running Background Tasks on the Server

There is a mechanism for starting a background job from a Zen page and tracking its progress within the page. The method runs in the background but the Zen page does not wait for the method to return before allowing the user to continue to interact with the Zen application. This can be useful for cases where a page needs to initiate a long-running task and wants to provide some degree of feedback to the client while waiting for that task to complete. Running a background task prevents a premature timeout of the Zen page, since the maximum configurable timeout for a Zen page may be less than the time required for some long-running tasks to complete. You must have a product license installed in order to use this Zen feature.

A Zen page class has a backgroundTimerInterval property. This is the interval, in milliseconds, at which timer events are fired to check on the status of background tasks started by this page. The default is 1000.

The following table describes the server-side methods that support background tasks; these are available in any class that inherits from %ZEN.Component.page.

Table 2–7: Page Class Server Side Methods for Background Tasks

Method	Description
%EndBackgroundMethod()	Clean up after a method that was invoked as a background task. This deletes any status information that was being maintained for the background task. This method is final. You can cannot override it.

Method	Description
%OnFinishBackgroundTask(*id*)	You can override this server-side callback method to control the behavior of the page when the client checks on the progress of a background task and the task is complete. Typically you would use this method to send back JavaScript to update the page, for example with a completion message.
%OnMonitorBackgroundTask(*id*,*status*,*val*)	You can override this server-side callback method to control the behavior of the page each time the client checks on the progress of a background task and the task is still running. Typically you would use this method to send back JavaScript to update the progress bar on the client. The following arguments are automatically provided to the callback when it is invoked: *id* is a string giving the job ID for the background task, *status* is the current job status as a string, and *val* is a %Float value that indicates how much of the background task is complete (as a percentage value between 0 and 100).
%RunBackgroundMethod(*method*,*arg1*,*arg2*,...)	Starts a background job to run a class method of this Zen page. **%RunBackgroundMethod**() returns a %Status value and may have one or more arguments: • *method* gives the name of the class method to run. • Depending on the method signature, zero or more arguments may follow — *arg1*, *arg2*, and so forth — these are the arguments of the *method* named in the first argument. Zen monitors only one background task at a time. If **%RunBackgroundMethod**() is called while a previous background task is running, the new method becomes the current monitored task. The previous task runs to completion, but the client is not notified about it. This method is final. You can cannot override it.

Method	Description
%SetBackgroundMethodStatus(*msg*,*val*)	Can be called from within a method that is running in the background, to update its own status information. **%SetBackgroundMethodStatus** has no return value and takes two arguments: • A %String specifies the status message seen by the client page. The string may be empty. • An optional %Float value indicates how much of the background task is complete (as a percentage value between 0 and 100). A client page can use this information to display progress to the user. This method is final. You can cannot override it.

2.3.4 Server Side Callback Methods for Components and Pages

The following table lists and describes server side callback methods supported by components (including the page itself). These are methods which, if defined, are called in response to specific events. These methods are available for you to implement or override, for example if you are writing custom components.

Table 2–8: Component and Page Server Side Callback Methods

Callback Method	Description
%OnDrawEnclosingDiv()	This server-side callback method must return a string that contains additional attributes for the HTML element that encloses this component on the display page. Be sure to start the returned string with a space character to prevent the new **%OnDrawEnclosingDiv**() attributes from conflicting with other attributes of the same .

2.4 Client Side Methods

The topics in this section describe important page and component methods that run on the client side.

2.4.1 Client Side Callback Methods for the Page

The client-side page object `zenPage` inherits a number of callback methods; the following table lists and describes them. These are JavaScript methods which, if defined, are called in response to specific events. These methods are available for you to implement or override, for example if you are writing custom components.

Table 2–9: Page Class Client Side Callback Methods

Callback Method	Description
onkeydownHandler(*evt*)	This client-side page method, if present, is fired when a keydown event occurs on the page. *evt* is a `zenEvent` object, a direct analog for the DOM `window.event` data structure. For details, see zenEvent in the section "Client Side Functions, Variables, and Objects."

Callback Method	Description
onkeyupHandler(*evt*)	This client-side page method, if present, is fired when a keyup event occurs on the page. *evt* is a `zenEvent` object, a direct analog for the DOM `window.event` data structure. For details, see zenEvent in the section "Client Side Functions, Variables, and Objects."
onlayoutHandler(*load*)	This client-side page method, if defined, is called when the page is first loaded or whenever it is resized. If this is called at load time, then *load* is true; otherwise it is false.
onloadHandler()	This client-side page method, if defined, is called when the client page is finished loading. It is triggered by the page's HTML onload event.
onlogoutHandler()	If this client-side page method is defined and the AUTOLOGOUT parameter for this page is set to 1 (true), this method is invoked when the logout timer for this page fires. Subsequent behavior depends on the return value of this method. • If this method returns true, the normal page logout behavior fires. That is, the page is reloaded, causing a login page to appear if the current session has ended. • If this method returns false, Zen does whatever this method specifies and then suppresses the normal page logout behavior.
onoverlayHandler(*index*)	This client-side page method, if present, is fired when a component with an overlay is clicked on. The *index* value is the system-assigned index number used internally to identify this component. For components within repeating groups, *index* 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 *index* value.
onPopupAction(...)	This client-side component method, if present, is fired when a Zen popup window has specified this page as its parent and subsequently fires an action. The page is the default parent of a popup window that has no parent component specified. See "Popup Windows" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.
onresizeHandler()	This client-side page method, if defined, is called when the client page is resized. It is triggered by the page's HTML onresize event.
onServerMethodCall(*method*)	If implemented, this client-side page method is called just before a server method in the page class is invoked. *method* is the method name. For details and examples, see "Notifying the Client When Server Side Methods Run."
onServerMethodError(*err*)	If implemented, this client-side page method is called whenever a server method call returns an error from the server. *err* contains the error message. For details and examples, see "Notifying the Client When Server Side Methods Run."
onServerMethodReturn(*method*)	If implemented, this client-side page method is called just after a server method in the page class is processed. *method* is the method name. For details and examples, see "Notifying the Client When Server Side Methods Run."

Callback Method	Description
onunloadHandler()	This client-side page method, if defined, is called when the client page is about to unload. It is triggered by the page's HTML onbeforeunload event. If the **onunloadHandler** method returns a string value, a confirmation dialog appears whenever the user attempts to navigate away from or refresh the page. The dialog displays the string that you specify as the return value of the **onunloadHandler** method. If the **onunloadHandler** method is not present, has no return statement, has no return value, returns true, returns false, or returns any non-string value, then no confirmation dialog appears when the user leaves or refreshes the page.

2.4.2 Notifying the Client When Server Side Methods Run

A Zen page can be notified before and after a server method is invoked. To be notified before a server method is called, a Zen page class should implement the abstract method **onServerMethodCall**, a client-side JavaScript method. For example:

Class Member

```
ClientMethod onServerMethodCall(method) [ Language = javascript ]
{
    alert('Call: ' + method);
}
```

To be notified just after a server method returns code to the client, a Zen page should implement the abstract method **onServerMethodReturn**, a client-side JavaScript method. For example:

Class Member

```
ClientMethod onServerMethodReturn(method) [ Language = javascript ]
{
    alert('Return: ' + method);
}
```

To be notified when a server method returns an error, a Zen page should implement the abstract method **onServerMethodError**, a client-side JavaScript method. For example:

Class Member

```
ClientMethod onServerMethodError(error) [ Language = javascript ]
{
    alert('Return: ' + error);
}
```

2.4.3 Client Side Methods for the Page

The client-side page object `zenPage` inherits a number of JavaScript methods that can be useful in client-side programming. Unlike the methods described in the previous section, "Client Side Page Callback Methods," these methods are final and cannot be overridden.

The following table lists the general purpose client-side methods. For a list of the client-side page class methods that allow you to modify the component object model, see the section "Defining Page Contents Dynamically on the Client."

Table 2–10: Page Class Client Side Methods

Method	Description
cancelPopup()	Close the current popup window with no further action. See "Popup Windows" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.
endModal()	Hide the current modal group, if there is one. See "Modal Groups" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.
fireOnLoadEvent()	Fire the onload event for every component on the page that defines one. Zen fires the events in reverse order according to the internal hierarchy of the page, so that it fires the page's onload event last.
fireOnResizeEvent()	Fire the onresize event for the page, if it defines one.
fireOnUnloadEvent()	Fire the onunload event for every component on the page that defines one. If any component's onunload handler returns a string value, Zen uses that as the return value of the page's onbeforeunload handler.
firePopupAction(...)	Notify the parent window of this popup that a user action has occurred. For full information, see "Popup Windows" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.
getComponent(*index*)	Find a component on the page given its *index* number, a system-assigned number used internally to identify this component. For components within repeating groups, *index* 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 *index* value.

getComponentById(*id*,*tuple*)	Find a component on the page given its *id* value. For components within repeating groups, the optional *tuple* number indicates the (1–based) position of this component within the group.
	The zen JavaScript function is a shorthand equivalent for **getComponentById**() when there is no *tuple* involved. You can use:
	`zen(id)`
	In place of:
	`zenPage.getComponentById(id)`

gotoPage(*url*)	Set the location of the browser to the specified new URI. Use **gotoPage** to navigate to new pages to ensure that URIs are encoded correctly.
launchPopupWindow(...)	Launch a popup window. For full information, see "Popup Windows" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.
setComponentId(*obj*,*id*)	Given the component identified by *obj*, change its *id* value.
setTraceOption(*name*,*flag*)	This client method lets you turn the client-side tracing flags on or off. *name* is the name of the tracing option. The *name* value can be 'events' (trace client events), 'js' (display the JavaScript returned from the server), or 'serialize' (display object serializations). *flag* is true to enable the specified option, false to disable it.

Method	Description
startModal(*obj*)	Make a component visible as a modal group. Alternately, use the **show** method of the <modalGroup> component. See "Modal Groups" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.

2.4.4 Client Side Callback Methods for Components and Pages

Components (including the page itself) inherit a number of callback methods; the following table lists and describes them. These are JavaScript methods which, if defined, are called in response to specific events. These methods are available for you to implement or override, for example if you are writing custom components.

Table 2–11: Component and Page Client Side Callback Methods

Callback Method	Description
onPopupAction(...)	If present, fires when a Zen popup window has specified this component as its parent and subsequently fires an action. For details, see "Popup Windows" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.
onEndModalHandler(*zindex*)	If present, fires upon notification that this component is about to stop being modal. That is, it is no longer 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. For details, see "Modal Groups" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.
onRefreshContents()	If present, this callback is invoked by the **refreshContents** method just after the new HTML is delivered to it from the server.
onStartModalHandler(*zindex*)	If present, fires upon notification that this component is about to become modal. That is, it is be 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. For details, see "Modal Groups" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.

2.4.5 Client Side Methods for Components and Pages

Every component on the page (including the page itself) inherits a number of useful JavaScript methods. The following table lists these methods. Specific components also have specialized methods not listed in the table. Unlike the methods described in the previous section, "Client Side Page Callback Methods," these methods are final and cannot be overridden. You can call them from your page class or custom component class as specified here.

Table 2–12: Component and Page Client Side Methods

Method	Description
findElement(*id*)	Get the HTML element associated with this component. *id* is the *id* value that you supplied for this component in the Zen page class.
getEnclosingDiv()	Get the HTML div element used to enclose a specific component.
getLabelElement()	Get the HTML element that displays the label for this component.

Method	Description
getHidden()	Return true if a component is currently hidden, otherwise false.
getHintElement()	Get the HTML element that displays the hint text for this component.
getProperty(*name*, *key*)	Return the value of the component property identified by *name*. The optional parameter *key* is useful when properties (such as collections) need a key to find a specific value.
getSettings(*settings*)	Get the set of named properties for this component. **getSettings** returns a list of property names in the associative array that is the argument for the method (*settings*). You can then retrieve and modify property values using these property names as arguments to **getProperty** and **setProperty**.
getValue()	(Subclasses of control only). Get the value of a control. This is equivalent to calling **getProperty**('value') on this component.
invokeSuper(*method*, *args*)	Invoke the super class implementation of the identified *method*. Supply the arguments for the method as a JavaScript array in *args*.
isOfType(*type*)	Test if a component is a subclass of (is a type of) another component. For the input argument *type*, use the simple name of a component class, such as form or control.
refreshContents(*flag*)	Make a request to the server to regenerate the HTML that defines this component. If *flag* is supplied and is set to true, and if deferred mode is not in effect, refresh the component synchronously. **refreshContents** has the effect of redisplaying the component on the browser page. For a <tablePane> component, use **executeQuery**() instead.
setHidden(*flag*)	If *flag* is supplied and is set to true, sets the component to be hidden and invokes the onhide callback for this component. If *flag* is supplied and is set to false, sets the component to be displayed and invokes the onshow callback.
setProperty(*name*, *val*)	Set the value of a property for a component. *name* gives the property name. *val* gives the value.
setPropertyAll(*name*, *val*)	Set the value of a property for a group component and all of its child components. *name* gives the property name. *val* gives the value. **setPropertyAll**() does not work for the disabled property; use **setProperty**() instead.
setValue(*val*)	(Subclasses of control only). Set the value of a control. This is equivalent to calling **setProperty**('value',*val*) on this component.
startProgressBar(*div*)	Start the display of a progress bar within the display area for this component. This can be useful for components that refresh their values from the server. If *div* is provided it is the enclosing HTML div that contains the progress bar.
stopProgressBar()	Stop the timer used by the progress bar.

2.5 Zen Properties on Client and Server

Any property defined within a Zen page class whose name does *not* start with % also becomes a property of the client-side page object. These client-side properties are initialized to the values they held on the server side, immediately before the page's **%DrawHTML** method was invoked.

If you want to provide values for these page properties before the page displays, you can provide initial values for page class properties in one of the following ways:

- Define an InitialExpression for the property in the page class definition, for example:

  Class Member

  ```
  Property dayList As %ZEN.Datatype.caption
    [ InitialExpression = "Sun,Mon,Tue,Wed,Thu,Fri,Sat" ];
  ```

- Set the property in the **%OnCreatePage** or **%OnAfterCreatePage** callback methods.

In server-side methods, you can get and set values of class properties directly, using normal dot syntax. For example:

Class Member

```
Method serverInstanceMethodCreate() [ ZenMethod ]
{
  #; create
  Set group = %page.%GetComponentById("group")
  If '$IsObject(group) {
    &js<alert('Unable to find group.');>
    Quit
  }
  Set count = group.children.Count()

  Set cal = ##class(%ZEN.Component.calendar).%New()
  Set cal.id = "NewCal"_count
  Do group.%AddChild(cal)

  Set btn = ##class(%ZEN.Component.button).%New()
  Set btn.id = "NewBtn"_count
  Set btn.caption = "New Button"
  Do group.%AddChild(btn)

  Set btn = ##class(%ZEN.Component.text).%New()
  Set btn.label = "Hey"
  Do group.%AddChild(btn)
}
```

To ensure proper synchronization of values between client and server, you should not access property values directly in client-side methods. You must use the get and set methods provided in the component classes. For single-valued properties, use the **getProperty** and **setProperty** methods. For example:

```
var index = table.getProperty('selectedIndex');
```

Or:

```
var svg = zen('svgFrame');
svg.setProperty('layout','');
svg.setProperty('editMode','drag');
```

Important: Do not confuse **getProperty** and **setProperty** with the **getValue** and **setValue** methods, which get and set the *value* of a control component, and do not apply to any other component property.

There is a limitation on the values that you can assign to the properties of Zen components. You cannot use values in the DOM that include any of the ASCII characters with numeric values 0 through 10. These characters are usually expressed in ObjectScript code using the $CHAR ($C) function with a numeric value, for example $C(1) or $C(4). These special

characters are reserved for use as delimiters within the Zen serialization code. Therefore, if you use any of the ASCII characters 0 through 10 in the values of ZEN component properties, these values cause errors when Zen deserializes the page.

Note that multidimensional properties are not supported as standard Zen page properties. You can achieve the same result using lists or arrays, or via JSON.

2.6 Zen Page Class Parameters

There are a number of class parameters that a page class can use to control the behavior of the page. This section lists the most useful of them. Others are described in the online class documentation.

APPLICATION

Package and class name of the associated application.

AUTOLOGOUT

If 1 (true) attempt to refresh this page when its session has expired. If 0 (false) do not. The default is 1.

If AUTOLOGOUT is 1, a Zen page generates JavaScript to set up a timer that reloads the page at about the time when the server session expires. It also generates code so that the client automatic logout timer is reset on every call to the server (the server already resets the session timeout). If AUTOLOGOUT is 0, then none of this code is generated.

If the **onlogoutHandler** client-side page method is defined and AUTOLOGOUT is set to 1 (true), **onlogoutHandler** is invoked each time the logout timer for this page fires. Subsequent behavior depends on the return value of the method, as follows:

- If this method returns true, the normal page logout behavior fires. That is, the page is reloaded, causing a login page to appear if the current session has ended.
- If this method returns false, Zen does whatever this method specifies and then suppresses the normal page logout behavior.

CSSINCLUDES

Comma-separated list of Cascading Style Sheet (.css) files for the page. You can use URLs or simple file names. If you use simple file names, the **CSP Files Physical Path** specifies the physical path to these files. For further information, see the section "Zen Application Configuration." For information about style sheets, see the "Zen Style" chapter in the book *Using Zen*.

DOMAIN

You must provide a value for this parameter if you wish to use Zen localization. The default is "" (an empty string indicating no domain).

HTMLATTRS

Zen appends this string to the generated HTML tags at the beginning of the output page, as a set of whitespace-delimited attributes. The default is "" (an empty string indicating that nothing should be appended).

JSINCLUDES

Comma-separated list of JavaScript (.js) include files for the page. You can use URLs or simple file names. If you use simple file names, the **CSP Files Physical Path** specifies the physical path to these files. For further information, see the section "Zen Application Configuration."

PAGENAME

Defines a text string that you can use in titles or labels. If not specified, the Zen page class name is used.

PAGETITLE

Default value for the <page> component's *title* attribute.

RESOURCE

Name of a system Resource for which the current user must hold USE privileges in order to view this page or to invoke any of its server-side methods from the client.

RESOURCE may be a comma-delimited list of resource names. In this case, the user must hold USE privileges on at least one of the given Resources in order to use the page.

For further details, see the section "Application Resources" in the "Assets and Resources" chapter of the *Caché Security Administration Guide*.

SHOWSTATS

If 1 (true) display server statistics (such as the time it took to process the page) within a comment at the end of the page. If 0 (false) do not. The default is 1.

USERPACKAGES

Comma-separated list of user-defined class packages whose HTML class and style definitions are in pre-generated include files.

USERSVGPACKAGES

Comma-separated list of user-defined class packages whose SVG class and style definitions are in pre-generated include files.

2.7 Zen Special Variables

Zen offers several special variables to represent instantiated objects of various types within the Zen application. These variables offer a convenient way to reference the methods and properties of these objects from within Zen page or Zen component class code.

Table 2–13: Special Variables on the Server Side

Server Side Variable Name	Refers To	Use in These Classes	Use in Zen Runtime Expressions
`%application`	Application object	Page	No
`%composite`	Current composite object (if any)	Page, Component	Yes
`%page`	Current page object	Page, Component	Yes

Server Side Variable Name	Refers To	Use in These Classes	Use in Zen Runtime Expressions
%query	Current ResultSet object (if any)	Page, Component	Yes
%session	Current CSP session object	Page, Component	Yes
%this	Current object (page or component)	Page, Component	Yes
%url	An object whose properties are URI parameters of the current page	Page, Component	Yes
%zenContext	String that indicates what the server code is currently doing	Page	No

Note that the %url special variable contains values only when a page is *first* served; If you refer to %url in subsequent method calls or component refreshes, its properties have no value. If you need to refer to a value passed in via a URI parameter, define a property of your page class, link it to a URI parameter using the ZENURL parameter, and then refer to this value via the %page object. For details, see the section "Zen Page URI Parameters."

Note: In general, local variable names starting with the % character are for system use only. Local variables that you define in your applications may begin with "%Z" or "%z"; all other percent variables are reserved for system use according to the rules described in "Rules and Guidelines for Identifiers" in the *Caché Programming Orientation Guide*.

2.7.1 %application

The server-side %application variable is available in every Zen page class. It is intended for use in methods that run while the page object is being created. It refers to the instance, on the server, of the application class that is associated with this page. This allows the page to invoke server-side methods defined in the application class.

For example, the following method is defined in the application class ZENDemo.Application in the SAMPLES namespace:

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
}
```

Page classes that are associated with this application can then invoke this method as follows:

Class Definition

```
Class ZENDemo.ExpenseCalculator Extends %ZEN.Component.page
{
  /// Application this page belongs to.
  Parameter APPLICATION = "ZENDemo.Application";

        // ...Intervening lines removed...

  /// Return an array of quick links to be displayed by the locator bar.
  ClassMethod GetQuickLinks(Output pLinks) As %Status
  {
    // dispatch to our application class
    Quit %application.GetQuickLinks(.pLinks)
  }
}
```

There is no client-side equivalent for `%application`.

2.7.2 %page and zenPage

Zen offers special variables to represent the Zen page object on the client and server sides:

- In ObjectScript, Caché Basic, or Caché MVBasic code that runs on the server side, the page object is `%page`:

  ```
  Set btn = %page.%GetComponentById("NewBtn1")
  ```

- In Zen runtime expressions, the page object is `%page`:

  ```
  text id="rows" label="Rows:" value="#(%page.Rows)#" size="5"/>
  ```

- In JavaScript methods that run on the client side, the page object is `zenPage`:

  ```
  var chart = zenPage.getComponentById('chart');
  ```

- In JavaScript expressions that are values of XML attributes, the page object is `zenPage`:

  ```
  <button caption="+" onclick="zenPage.makeBigger();"/>
  ```

2.7.3 %this, this, and zenThis

Zen offers special variables to represent a Zen object (component or page) on the client and server sides:

- In ObjectScript, Caché Basic, or Caché MVBasic code that runs on the server side, the object itself is `%this`:

  ```
  Set %this.Name = pSource.Name
  ```

- In Zen runtime expressions, the object itself is `%this`:

  ```
  <html>Item #(%this.tuple)#: <b>#(%query.Title)#</b></html>
  ```

- In JavaScript methods that run on the client side, the object itself is `this`:

  ```
  var out = this.getComponentById('events');
  ```

- In JavaScript expressions that are values of XML attributes, the object itself is `zenThis`:

  ```
  onchange="zenPage.changeLayout('svgFrame',zenThis.getValue());"
  ```

2.7.4 %zenContext

When any activity on the server is being initiated by Zen, the server-side special variable `%zenContext` tells you which type of activity it is. `%zenContext` is a string. If `%zenContext` has the value:

- `page`, Zen is serving the page in response to an HTTP request

- `submit`, Zen is processing a submit request
- `method`, Zen is processing a hyperevent

Sometimes it is possible to have trouble with code being executed at compile time. To detect this, check for:

`$G(%zenContent)==""`

There is no client-side equivalent for `%zenContext`.

2.8 Client Side Functions, Variables, and Objects

This section describes the client-side JavaScript utility functions, variables, and objects that are available for use in Zen page classes.

2.8.1 zen

`zen(`*id*`)`

Find a Zen component by *id* value. Returns the object that matches the input *id*.

The `zen(`*id*`)` JavaScript function is equivalent to the following client-side JavaScript method call on the page object:

`zenPage.getComponentById(`*id*`)`

You can use the `zen(`*id*`)` function wherever JavaScript syntax is appropriate; for example:

XML

```
<button caption="Save" onclick="zen('MyForm').save();"/>
```

2.8.2 zenEvent

`zenEvent.`*property*

Suppose you have:

- A Zen component such as <listBox>
- ...with an event handler attribute such as *onchange*
- ...whose value is a JavaScript expression
- ...that invokes a client-side JavaScript method defined in the page class
- ...such as **notifyOnChange** in the following <listBox> example:

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" />
</listBox>
```

Inside the client-side JavaScript method, such as **notifyOnChange**, it is convenient to be able to refer to the JavaScript object that represents the event itself (the mouse click or a key press). This object contains supplementary information that

you might want to use in processing the event. The `zenEvent` special variable is available for this purpose. Each `zenEvent.type` value is a text string; for example, your code might check the event type as follows:

```
if (zenEvent.type=="mousedown") {
  // do something
  }
```

The following JavaScript example retrieves the `zenEvent.target` value to determine the exact target of a mouse click that occurred within a group. The goal is to find out whether the click was on the group itself (which has an enclosing and therefore a `zen` attribute) or on a component within the group.

```
// look at source element; IE does not support standard target property.
var target = (null == zenEvent.target) ? zenEvent.srcElement : zenEvent.target;

// all enclosing divs will define an attribute called 'zen'.
var zen = target.getAttribute('zen');
if (zen) {
    // do something
    }
```

The `zenEvent` object is a direct analog for the DOM `window.event` data structure. The following properties are available on the `zenEvent` object:

- `zenEvent.clientX`
- `zenEvent.clientY`
- `zenEvent.returnValue`
- `zenEvent.srcElement`
- `zenEvent.target`
- `zenEvent.type` may have one of the following values.

  The W3C has defined the type, scope, and handling for these events; however, support for this standard varies from browser to browser. Firefox is generally compliant in its implementation. Internet Explorer implements a commonly used subset of the standard, provides functional alternatives for many parts of the standard that they do not support, and continues to support a wide variety of proprietary legacy events that predate the W3C recommendation. Developers seeking cross-platform support should confine their designs to the events in the following list:

  - beforeunload
  - blur
  - change
  - click
  - contextmenu
  - dblclick
  - error
  - focus
  - keydown
  - keyup
  - keypress
  - load
  - mousedown
  - mouseup

- mousemove
- mouseout
- mouseover
- reset
- resize *
- select *
- submit
- unload

Note: For the events marked with an asterisk (*) in the previous list, the name and intent are standard across all browsers, but the scope, target context, and propagation rules are browser specific. Safari implements most of the standard events as well as several platform-specific extensions. Some versions of Safari do not natively recognize dblclick.

2.8.3 zenGetProp

`zenGetProp(`*id*`,`*prop*`)`

Returns the value of a named property within a Zen component, where:

- *id* is either the id of a component, or the component object itself.
- *prop* is the name of the property to set.

The `zenGetProp(`*id*`,`*prop*`)` JavaScript function is equivalent to the following ObjectScript method call:

`zenPage.getComponentById(`*id*`).getProperty(`*prop*`)`

For example, this JavaScript:

```
zenGetProp('ctrlButton','hidden');
```

Is equivalent to this ObjectScript:

```
zenPage.getComponentById('ctrlButton').getProperty('hidden');
```

2.8.4 zenIndex

`zenIndex(`*idx*`)`

Find a Zen component by the system-assigned index number of the component. For components within repeating groups, the index may include a dot followed by a 1-based number indicated the position of this component within the repeating group. Returns the object that matches the input *idx* number.

This JavaScript function is equivalent to the following ObjectScript method call:

`zenPage.getComponent(` *idx*`)`

2.8.5 zenInvokeCallbackMethod

`zenInvokeCallbackMethod(`*attr*`,`*ptr*`,'`*name*`',`*arg1*`,`*val1*`,`*arg2*`,`*val2*`,`*arg3*`,`*val3*`)`

A utility function used by components to invoke a callback method. The full list of **zenInvokeCallbackMethod** arguments is as follows. The first three arguments are required; the others are optional:

- A callback attribute of the component object, for example `this.onclick` or `this.onchange`. The value of this attribute must be a JavaScript expression that invokes a client-side JavaScript method.
- A pointer to the component object, for example `this`
- The HTML name for the event (must be quoted)
- If an optional fourth argument is provided, it is the name of a formal parameter to be passed to the callback method.
- If there is a fourth argument, a fifth argument must provide a value for the formal parameter.
- **zenInvokeCallbackMethod** permits up to two more pairs of argument names and argument values. These are the optional sixth, seventh, eighth, and ninth arguments for **zenInvokeCallbackMethod**.

For example:

```
zenInvokeCallbackMethod(this.onclick,this,'onclick')
```

For more information about this example, see the "Attaching Event Handlers to HTML Elements" section in the chapter "Custom Components."

2.8.6 zenLaunchPopupWindow

Deprecated. Use a client-side method that calls the Zen page method **launchPopupWindow** instead. For full information, see "Popup Windows" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.

2.8.7 zenLink

`zenLink(`*uri*`)`

A function that escapes any special characters found in a string that is intended to be used as a URI. *uri* must be a valid string. Enclose literal strings in single quotes, for example:

```
zenLink('MyApp.MyDialog.cls')
```

Also see the section "Popup Windows" in the "Popup Windows and Dialogs" chapter of *Using Zen Components*.

2.8.8 zenPage

`zenPage.`*property*

`zenPage.`*method*`(`*parameters*`)`

zenPage is the client side equivalent of %page. See "%page and zenPage" in the section "Zen Special Variables."

2.8.9 zenSetProp

`zenSetProp(`*id*`,`*prop*`,`*value*`)`

Set the value of a named property within a Zen component, where:

- *id* is either the id of a component, or the component object itself.
- *prop* is the name of the property to set.
- *value* is the value to assign to the property.

The `zenSetProp(`*id*`,`*prop*`,`*value*`)` JavaScript function is equivalent to the following ObjectScript method call:

`zenPage.getComponentById(`*id*`).setProperty(`*prop*`,`*value*`)`

For example, this JavaScript:

```
zenSetProp('ctrlButton','hidden', false);
```

Is equivalent to this ObjectScript:

```
zenPage.getComponentById('ctrlButton').setProperty('hidden',false);
```

2.8.10 zenSynchronousMode

`zenSynchronousMode`

A flag with the value `true` (all methods are executed synchronously regardless of whether or not they have return values) or `false` (methods are synchronous if they have a return value, asynchronous otherwise). See the section "Synchronous and Asynchronous Methods."

2.8.11 zenText

`zenText(`*id*`,`*p1*`,`*p2*`,`*p3*`,`*p4*`)`

zenText is the client side equivalent for $$$Text. See "Localization for Client Side Text" in the chapter "Zen Localization."

2.8.12 zenThis

`zenThis.`*property*

`zenThis.`*method*`(`*parameters*`)`

zenThis is the client side equivalent for %this. See "%this, this, and zenThis" in the section "Zen Special Variables."

2.9 Zen Runtime Expressions

Rather than always providing static information in a Zen page description, sometimes it is useful for the page running on the client to be able to invoke runtime expressions that execute on the server. For example, perhaps the caption text for a button needs to be different depending on the identity of the currently logged-in user, which can only be determined from the server at runtime. Various runtime values can play a part in these decisions. For this reason, Zen provides a way for client-side application code to contain expressions that are resolved only on the server, and only at runtime.

Important: The syntax rules for Zen runtime expressions are extremely specific.

A runtime expression is enclosed within `#( )#` and looks something like this:

XML

```
<button caption="#(myObject.myProperty)#" />
```

Where `myProperty` has a specific value on the server at runtime.

Zen runtime expression syntax can be used, *with restrictions*, in the following contexts only:

- XML that is embedded within an XData block (certain XML elements and attributes only)
- JavaScript that is embedded within a &js<> block in a ZenMethod
- A JavaScript expression provided as the value of an XML attribute such as *onclick*
- HTML that is embedded within an &html<> block in a server-side method

The following items cannot appear within the #( )# delimiters of a Zen runtime expression:

- Methods, such as %this.method(). Only properties are allowed.
- Properties at a level greater than first level. For example, chaining such as this.parent.other.thing. is not allowed.
- Calls to macros, such as $$$Text.
- JavaScript expressions.
- ObjectScript expressions. The exception to this rule is when a runtime expression appears inside a server-side ObjectScript method or ZenMethod. See "Runtime Expressions in Server-Side Methods."

2.9.1 Runtime Expressions in XML

Zen runtime expressions can be used within a Zen class XData block, but only the values of specific XML elements and attributes can contain Zen runtime expressions. One such example is the <button> *caption* attribute:

XML

```
<button caption="#(%query.Name)#" onclick="alert('#(%query.Name)#')"/>
```

Of all the possible <button> attributes, only *caption*, *value*, and *hidden* support Zen runtime expressions. The value of *onclick* in the above example does contain a Zen runtime expression, but this is acceptable because the value of *onclick* is a JavaScript expression, which may contain #( )# syntax. Refer to the restrictions listed in the topic above.

XML elements and attributes that support Zen runtime expressions are clearly identified throughout this book; you may find them by searching for the #( )# string in the book text.

Another way to tell that an XML attribute supports runtime expressions is to consult the Caché class documentation to see if the corresponding property has the datatype parameter ZENEXPRESSION set to 1 (true).

You cannot cause a property to support runtime expressions by setting ZENEXPRESSION=1. Built-in Zen component classes provide ZENEXPRESSION=1 to *indicate* that the property supports runtime expressions, not to enable it to do so.

On the other hand, when you create your own component subclasses, you can set ZENEXPRESSION=1 for properties in those subclasses. It must be a property of a custom component class and not a property of one of the built-in Zen component classes. For details, see the "Datatype Parameters" section in the chapter "Custom Components."

A significant number of XML attributes support Zen runtime expressions, but only one XML element supports them. This is the <html> component, which may contain literal text and #( )# runtime expressions. For example:

XML

```
<html>Item #(%this.tuple)#: <b>#(%query.Title)#</b></html>
```

2.9.2 Runtime Expressions and Special Variables

A runtime expression may use the server-side special variables listed in the section "Zen Special Variables." Zen runtime expressions support the following server-side special variables *only*:

- %composite
- %page
- %query
- %session

- %this
- %url

By limiting expressions to a predefined set of object property values, the page developer maintains control over which server-side values the client can see. Otherwise it would be possible for a page to execute code that the page developer did not intend, and thus present a security risk.

For example, the %session you see in a Zen expression is *not* the %session object from the server side. It is a client-side expression that can be evaluated to get a user-data value from the %session object. Suppose your page has a component whose *id* is test. If your Zen expression uses:

```
#(%session.test)#
```

Then on the server this becomes:

```
%session.Data("test")
```

And the value associated with the component whose *id* is test is returned in place of the Zen expression.

For more information about the %session object, see the "%CSP.Session Object" section in the book *Using Caché Server Pages (CSP)*.

2.9.3 Runtime Expressions in Server-Side Methods

The following is an example of a server-side method with an &html<> block, in which a runtime expression invokes the ObjectScript function $ZCONVERT ($ZCVT) to help it to write out the elements of a bulleted list:

Class Member

```
Method %DrawHTML()
{
  &html<<div class="demoSidebar">>
  Write $ZCVT(..text,"O","HTML")

  #; bullets
  Set tCount = ..bullets.Count()
  If (tCount > 0) {
    &html<<ul>>
    For n = 1:1:tCount {
      Set tBullet = ..bullets.GetAt(n)
      &html<<li>#($ZCVT(tBullet.text,"O","HTML"))#</li>>
      }
    &html<</ul>>
    }
  }
  &html<</div>>
}
```

The following is an example of a ZenMethod with a &js<> block, in which a runtime expression passes the server-side identifier *pIndex* to JavaScript code that needs to execute on the client. In a later &js<> block, runtime expressions invoke the ObjectScript function $RANDOM ($R) to produce drawing coordinates, and provide CSS style statements to configure a graphical object:

```
ClassMethod GetSVGContents(pIndex As %Integer) [ ZenMethod ]
{
  #; get the svg component
  &js<var svg = zenPage.getComponent(#(pIndex)#);>

  #; lines
  For i=1:1:30 {
    &js<
      var line = svg.document.createElementNS(SVGNS,'circle');
      //line.setAttribute('x1',200);
      //line.setAttribute('y1',100);
      line.setAttribute('r',5);
      line.setAttribute('cx',#(10+$Random(380))#);
      line.setAttribute('cy',#(10+$Random(180))#);
      line.setAttribute('style',
```

```
        '#("fill: yellow; stroke: black; stroke-width: 2;")#');
      svg.svgGroup.appendChild(line);
    >
    }
  Quit
}
```

Note: The samples in this topic demonstrate how runtime expressions can be used in a server-side method. In a client-side JavaScript method, you can achieve similar results without using runtime expressions.

2.10 Zen Proxy Objects

It can be convenient to have a simple way to pass arbitrary data between the client page and a server process. For this reason, Zen defines a simple generic object called a *proxy object*. You can:

1. Create this object within a client page
2. Pass it to the server (where it is automatically marshalled as a server object)
3. Use and modify it on the server
4. Pass values to the client

This section describes this sequence and provides syntax examples.

2.10.1 Proxy Objects on the Client

On the client, a proxy object is simply an instance of a JavaScript `zenProxy` object. You can create a `zenProxy` object and fill in its properties as you would any other object; for example:

```
var proxy = new zenProxy();
proxy.Name = 'John Smith';
proxy.City = 'Boston';
```

2.10.1.1 Set Properties

A proxy object is completely generic; it has no predefined properties. You can set whatever property value you wish within a zenProxy object, subject to the following restrictions:

- The value of a property must be a literal value (string or number) and not an object or function. Only literal values are supported; you cannot set the value of a zenProxy property to be an object. If you need a more complex object like this, build a normal Zen object.
- The name of a property must be valid in JavaScript.
- A zenProxy object cannot be a property of a Zen component. If you use the following syntax in a Zen component class, it fails to compile:

  Class Member

  ```
  Property a As %ZEN.proxyObject;
  ```

  You may subclass %ZEN.Component.object instead.

2.10.1.2 Clear Properties

You can remove all properties within a zenProxy object by using its clear method:

```
obj.clear();
```

2.10.2 Passing a Proxy Object to the Server

You can pass a proxy object to the server by defining a ZenMethod and declaring one of its arguments to be of type %ZEN.proxyObject. For example:

Class Member

```
ClassMethod MyMethod(pProxy As %ZEN.proxyObject) As %Boolean [ZenMethod]
{
  Set x = pProxy.Name
  Quit 1
}
```

2.10.3 Proxy Objects on the Server

When the client invokes **MyMethod** (as defined in the previous topic) it passes an instance of the zenProxy object to the server. On the server, this instance is automatically converted to a %ZEN.proxyObject with all the same properties as the client object.

The server side %ZEN.proxyObject object is also completely generic, with no predefined properties.

Important: Do not attempt to use the $Data or $Get function with an instance of %ZEN.proxyObject. If you do so, the function returns an "Invalid Property" error.

2.10.3.1 Get Properties

You can retrieve values from the %ZEN.proxyObject object using normal property syntax. For example:

```
 Set a = pProxy.property1
 Set b = pProxy.property2
```

You can also get the complete set of properties within a %ZEN.proxyObject object as a local array using the **%CopyToArray** method:

```
 Do pProxy.%CopyToArray(.array)
```

If you refer to a property that is not currently defined, you get a value of "" and not an error.

2.10.3.2 Set Properties

Setting a property automatically defines a property if it is not already defined:

```
 Set pProxy.property3 = "value"
```

There is also a **%CopyFromArray** method to set the complete set of properties within a %ZEN.proxyObject object.

2.10.3.3 Clear Properties

You can remove all properties from the server side %ZEN.proxyObject object using its **%Clear** method:

```
 Do pProxy.%Clear()
```

2.10.4 Passing Values to the Client

If a ZenMethod that is passed a %ZEN.proxyObject object modifies it, the changes are automatically applied to the client version of the object when the method call completes. This includes any properties that have been added or removed.

Important: If your ZenMethod has no return value, then it is executed asynchronously. This is Zen normal behavior; for details, see the section "Synchronous and Asynchronous Methods." In this case, changes to the client-side zenProxy object are still applied, but you do not know *when* these changes are applied.

It is also possible to define a ZenMethod that returns a %ZEN.proxyObject:

Class Member

```
ClassMethod MyCreate() As %ZEN.proxyObject [ZenMethod]
{
  Set tProxy = ##class(%ZEN.proxyObject).%New()
  Set tProxy.Data = "Some data from the server"
  Quit tProxy
}
```

A client method can call this ZenMethod as follows:

Class Member

```
ClientMethod clientMethod() [Language = JavaScript]
{
  var proxy = zenPage.MyCreate();
  alert(proxy.Data);
}
```

2.10.5 Proxy Object Examples

This section describes built-in Zen components that offer opportunities to use proxy objects.

2.10.5.1 <html>

The <html> component provides an **OnDrawContent** callback that can supply the content of the component. This callback is passed a *seed* value: an application-specific string that is interpreted by the callback method. If the client sets this *seed* value to a zenProxy object, then it is automatically marshalled as a **%ZEN.proxyObject** on the server when the **OnDrawContent** callback is called.

Suppose a Zen page defines an <html> component as follows:

XML

```
<html id="html" OnDrawContent="DrawHTML"></html>
```

And also defines **OnDrawContent** as follows. This **OnDrawContent** method checks to see if the input argument *pSeed* is an object, and if so it retrieves and displays its Name and SSN properties. Otherwise, it displays an error message:

Class Member

```
ClassMethod DrawHTML(pSeed As %String) As %Status
{
  If $IsObject(pSeed) {
    &html<<table>>
    &html<<tr><td>Name:</td><td>#(pSeed.Name)#</td></tr>>
    &html<<tr><td>SSN:</td><td>#(pSeed.SSN)#</td></tr>>
    &html<</table>>
  }
  Else {
    &html<<i>No data to display</i>>
  }
  Quit $$$OK
}
```

You could update this <html> component from a client side JavaScript method by instantiating a new zenProxy object, assigning values to Name and SSN properties on that object, assigning this zenProxy object as the value of the *seed* property for the <html> component, and then refreshing the component, as in the following JavaScript example:

Class Member

```
ClientMethod updateHTML() [Language = JavaScript]
{
  // find the html component
  var html = zen('html');
  if (html) {
    var proxy = new zenProxy();
    proxy.Name = 'John';
    proxy.SSN = '222-22-2222';

    // set seed value and invoke server refresh
    html.setProperty('seed',proxy);
    html.refreshContents();
  }
}
```

For more about the <html> component, see its description in the "Other Zen Components" chapter of *Using Zen Components*.

2.10.5.2 <dataController>

The <dataController> component has a method, **getDataAsObject**(), that returns the current set of properties within a <dataController> as a zenProxy object. This provides an easy way to get data from a controller and pass it to the server for some purpose other than those already built into the <dataController>.

For more about the <dataController> component, see the "Data Controller" section of the "Model View Controller" chapter in *Using Zen Components*.

2.10.5.3 <form> and <dynaForm>

The <form> and <dynaForm> components have a method, **getValuesAsObject**(), that returns the current values of all controls in this form as a zenProxy object. The names of the properties within the proxy object are based on each control's *name* attribute.

For more about the <form> and <dynaForm> components, see the "Zen Forms" chapter in *Using Zen Components*.

2.11 Zen JSON Components

The Zen <altJSONProvider> component provides a way to transport object data between a server and client, or between client and server, using JavaScript Object Notation (JSON) format. Zen also provides a specialized version of the <altJSON-Provider>, which is discussed in the subsection "altJSONSQLProvider."

Note: The Zen component <altJSONProvider> is the replacement for <jsonProvider>, which is no longer specifically described. The <altJSONProvider> component has better performance and can be used in the same way as the <jsonProvider> component. Similarly, <altJSONSQLProvider> is the replacement for <jsonSQLProvider>.

2.11.1 Introduction

JSON refers to a JavaScript programming technique that allows you to define a set of one or more objects using object literal syntax. For example:

```
var obj = {name:'Bill', home:'New York'};
```

The Zen <altJSONProvider> component works as follows:

- Place an instance of the invisible <altJSONProvider> component in your <page> definition in XData Contents.
- Supply a value for the <altJSONProvider> *OnGetTargetObject* attribute that is the name of a method in the page class that creates an object or set of objects and returns it. The returned object can be an instance of a specific class or classes, or it can use the Zen proxy object class, %ZEN.proxyObject.
- Write the *OnGetTargetObject* callback method. See "The OnGetTargetObject Callback Method."
- When the page is rendered, the <altJSONProvider> converts the target object to a set of JavaScript objects. You can see these objects if you view the source of the page as sent to the client.
- The <altJSONProvider> has a client-side method, **getContentObject**(), which returns the client-side version of the target object. This is a graph of generic JavaScript Object objects that have the same properties and values as the target objects. If the target object refers to other objects or has collections (literal or object-valued) then the JavaScript object has corresponding object or collection properties. The client can modify these client-side objects or replace them completely using the **getContentObject**() method.
- The client can ship its content objects back to the server for processing by calling the **submitContent**() method. This converts the objects back into server-side objects and invokes the page class method specified by the <altJSONProvider> *OnSubmitContent* property, if there is one.
- If you are using *OnSubmitContent*, write the corresponding method in the page class (see details below). The *OnSubmitContent* callback method can modify the objects shipped to it or return a completely different set of objects. This makes it possible to use <altJSONProvider> as a way to execute different types of server operations.
- Alternative actions are available depending on the attributes you choose for the <altJSONProvider> element in XData Contents, and how you write your callback methods in the Zen page class.

Using the <altJSONProvider> component as an object transport has advantages and disadvantages when compared with other mechanisms provided by Zen (such as the built-in transport provided for Zen components). The main advantage is that you can transport data without having to create or modify server classes. You can ship almost any server-side object using this technique. The disadvantages are:

- You can ship a set of objects, but the objects must form a graph from a parent object down through levels of children; this is due to how JSON format data is reconstituted on the client. You cannot have child objects refer to parents, siblings or other objects outside of the graph.
- This approach uses late binding, so it is not as efficient as the code-generated approach used by Zen components.
- Not all object properties are supported: you cannot ship streams or binary values. Only references to child objects are transported.

An <altJSONProvider> element may contain zero or more <parameter> elements. Each <parameter> is passed as an argument to the callback method identified by *OnGetTargetObject*, *OnGetArray*, or *OnRenderJSON*, depending on which is being used to populate the JSON format object. These <altJSONProvider> attributes are mutually exclusive, so only one of

them—*OnGetTargetObject*, *OnGetArray*, or *OnRenderJSON*—actually uses the <parameter> values supplied with the <altJSONProvider>.

The <parameter> element has the following attributes.

Attribute	Description
paramName	The *paramName* must be unique within the <altJSONProvider>. 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 section "The OnGetTargetObject Callback Method." provides an example using <parameter>. The syntax is the same for the other callbacks. For additional examples of using <parameter> elements with Zen components, see the sections "Data Sources" and "Query Parameters" in the "Zen Tables" chapter of *Using Zen Components*.

2.11.2 JSON Provider Properties

As an XML element in XData Contents, <altJSONProvider> has the attributes listed in the following table.

Attribute	Description
OnGetArray	The name of a callback method defined in the page class. This method is automatically invoked when the page containing this <altJSONProvider> element is rendered. See "The OnGetArray Callback Method."
OnGetTargetObject	The name of a callback method defined in the page class. This method is automatically invoked when the page containing this <altJSONProvider> element is rendered. See "The OnGetTargetObject Callback Method."
OnRenderJSON	The name of a callback method defined in the page class. This method is automatically invoked when the page containing this <altJSONProvider> element is rendered. See "The OnRenderJSON Callback Method."
OnSubmitContent	The name of a callback method defined in the page class. This method is invoked when the client submits an object to the server by calling the client-side method **submitContent**(). See "The OnSubmitContent Callback Method."
targetClass	Class name of the target object expected to be served by this <altJSONProvider> component. You can dynamically set the *targetClass* value from server-side code by calling the **%SetTargetObject**(*obj*) method, where *obj* is a pointer to a %RegisteredObject.

Several of these attributes override each other, as follows:

- The *OnGetTargetObject* callback method provides the view of the object.
- If an *OnGetArray* callback is defined, the page invokes the *OnGetArray* callback to get the view, instead of the *OnGetTargetObject* callback.
- If an *OnRenderJSON* callback is defined, the activities in the *OnRenderJSON* callback override all of the default behavior of the <altJSONProvider> component when it is rendered. This means the callbacks identified by *OnGetTargetObject* or *OnGetArray* are not invoked. The *OnRenderJSON* callback does all of the work itself.

2.11.2.1 The OnGetArray Callback Method

The method identified by the *OnGetArray* property provides an easy way to ship a set of identical objects to the client by filling in a multidimensional array.

The following example shows the required signature for the method identified by *OnGetArray*:

Class Member

```
Method GetArray(ByRef pParameters,
                Output pMetaData,
                Output pData) As %Status
{
  Set pMetaData = $LB("name","rank","serialNo")
  Set pData(1) = $LB("Smith","Captain","444-33-2222")
  Set pData(2) = $LB("Jones","Corporal","333-22-3333")
  Quit $$$OK
}
```

Where:

- *pParameters* is an array of <parameter> values subscripted by *paramName* . <parameter> values are optional in an <altJSONProvider> definition. You only need to specify them if the callback method requires them. See *OnGetTargetObject* for a <parameter> example.
- *pMetaData* is a $List containing the names of the properties of the objects in the order in which they appear. The method must fill in this structure.
- *pData* is an array containing the data. Each node in the array should be a $List containing values for properties. This must match the metadata provided in pMetaData. The array of data can use any subscript value it wants. It is possible to define a hierarchical array. In this case, child nodes are placed within a parent collection called *children*. The method must fill in this structure.

In the above case the <altJSONProvider> definition would specify:

XML

```
<altJSONProvider OnGetArray="GetArray" />
```

The above example would result in two objects being shipped to the client in JSON format as follows:

```
var content = {
  name:'Smith',rank:'Captain',serialNo:'444-33-2222',
  children:[
    {
    name:'Jones',rank:'Corporal',serialNo:'333-22-3333'
    }
  ]
};
```

2.11.2.2 The OnGetTargetObject Callback Method

The following example shows the required signature for the method identified by *OnGetTargetObject*:

Class Member

```
Method GetTarget(ByRef pParameters,
       Output pObject As %RegisteredObject) As %Status
{
  Set pObject = ##class(MyApp.MyClass).%New()
  Set pObject.Name = "Bob"
  Set pObject.Department = pParameters("Dept")
  Quit $$$OK
 }
```

Where:

- *pParameters* is an array of <parameter> values subscripted by *paramName* . <parameter> values are optional in an <altJSONProvider> definition. You only need to specify them if the callback method requires them.
- *pObject* is an instance of the object whose data is to be provided to the client in JSON format. The method must return this object by reference. *pObject* can be any object that satisfies JSON restrictions as described in this topic. In fact, *pObject* can be a Zen proxy object as described in the "Zen Proxy Objects" section.

Important: InterSystems recommends that you do *not* ship a persistent object direct from the database in this fashion, especially if it has references to other objects, as you may end up shipping your entire database to the client. You should always use a simpler wrapper object in this case.

In the above case the <altJSONProvider> definition would specify:

XML

```
<altJSONProvider OnGetTargetObject="GetTarget" >
  <parameter paramName="Dept" value="Sales" />
</altJSONProvider>
```

2.11.2.3 The OnRenderJSON Callback Method

Note: This is an advanced callback for very special situations. It lets you write out the exact JSON content you wish to render to the client. You should be an expert in JavaScript debugging to use this feature.

The following example shows the required signature for the method identified by *OnRenderJSON*:

Class Member

```
Method RenderHandler(ByRef pParameters) As %Status
{
  /// Render activities here
  Quit tSC
}
```

Where:

- *pParameters* is an array of <parameter> values subscripted by *paramName* . <parameter> values are optional in an <altJSONProvider> definition. You only need to specify them if the callback method requires them. See *OnGetTargetObject* for a <parameter> example.

In the above case the <altJSONProvider> definition would specify:

XML

```
<altJSONProvider OnRenderJSON="RenderHandler" />
```

2.11.2.4 The OnSubmitContent Callback Method

The following example shows the required signature for the method identified by *OnSubmitContent*:

Class Member

```
Method SubmitHandler(pCommand As %String,
       pProvider As %ZEN.Auxiliary.altJSONProvider,
       pObject As %RegisteredObject,
       Output pResponse As %RegisteredObject) As %Status
{
  Set tSC = $$$OK
  If ($IsObject(pObject)) {
    Set tSC = pObject.%Save()
  }
  Quit tSC
}
```

Where:

- *pCommand* is the command string supplied to **submitContent**().
- *pProvider* is the JSON provider object.
- *pObject* is the submitted object after it has been converted from JSON format back into an object instance.
- If the callback method returns an object via the *pResponse* argument, this object is returned to the client and becomes the new content of the JSON provider.

In this case the <altJSONProvider> definition would specify:

XML

```
<altJSONProvider OnSubmitContent="SubmitHandler" />
```

2.11.3 JSON Provider Methods

A <altJSONProvider> element is the XML projection of the %ZEN.Auxiliary.altJSONProvider class. Objects of this class offer several methods that you can call to work with the component on the client side. The following table describes them.

Client Side Method	Description
getContentObject()	Return the client-side JSON data as an object, or if not available return null.
getError()	Get the current value of the error encountered on the server side. For details, see **submitContent**().
reloadContents()	Reload the contents of the JSON provider with data from the server. Unlike the **submitContent**() method, **reloadContents**() does not send data to the server. **reloadContents**() is typically used in conjunction with the *OnGetArray* callback: **reloadContents**() calls the server and the server, in turn, calls the *OnGetArray* callback to create new content to ship back to the client.
setContentObject(*obj*)	Make *obj* the new target object for this JSON provider.
setContentText(*json*)	Set the content for this provider using the string provided in the *json* argument. *json* is expected to contain object data in JSON format.

Client Side Method	Description
submitContent(*cmd*,*target*)	Send the current target object for this provider to the server for processing. This recreates the object on the server and invoke the *OnSubmitContent* callback method. **submitContent**() has the following arguments: • *cmd* is an optional string passed to the server callback method to allow for different behaviors in the server logic. • *target* is optional. If specified, it gives the name of the server-side class that you wish to have instantiated on the server. This has the same effect as setting the targetClass property of the JSON provider. The *target* argument makes it possible to submit content for different object classes. If the server cannot create an instance of the specified class, it records an error. The **submitContent**() method returns true if successful and false otherwise. If the method fails on the server, it records a string describing the error. This string can be retrieved on the client side using the **getError**() method.

2.11.4 altJSONSQLProvider

Note: The Zen <altJSONSQLProvider> component is the replacement for the <jsonSQLProvider> component, which is no longer specifically described. The <altJSONSQLProvider> component has better performance and can be used in the same way as the <jsonSQLProvider> component.

The <altJSONSQLProvider> component uses an SQL statement to supply data to the client. You can provide the SQL statement in the following ways:

- As the value of the *sql* attribute
- In the *OnGetSQL* callback method
- As a class query by providing *queryClass* and *queryName* attributes

The following code fragment illustrates use of the *sql* attribute.

XML

```
<page xmlns="http://www.intersystems.com/zen">
  <altJSONSQLProvider id="json"
    sql="SELECT ID,Name,SSN,DOB FROM Sample.Person" />
  <dataGrid id="grid" controllerId="json"/>
</page>
```

You can also supply parameters to the SQL statement, as illustrated in this example:

XML

```
<page xmlns="http://www.intersystems.com/zen">
  <altJSONSQLProvider id="jsonSQL"
    sql="SELECT ID,Name,Age FROM Sample.Person where Name %STARTSWITH ? or Age ?"  >
    <parameter paramName="1" value="Z"/>
    <parameter paramName="2" value="80"/>
  </altJSONSQLProvider>
  <dataGrid id="grid" controllerId="jsonSQL"/>
</page>
```

The next code fragment sets the *OnGetSQL* attribute of the <altJSONSQLProvider>, and provides parameters that are passed to the event handler.

XML

```
<page xmlns="http://www.intersystems.com/zen">
  <altJSONSQLProvider id="jsonSQL" OnGetSQL="GetSQL" >
    <parameter paramName="1" value="Z"/>
    <parameter paramName="2" value="80"/>
  </altJSONSQLProvider>
  <dataGrid id="grid" controllerId="jsonSQL"/>
</page>
```

This code sample shows an implementation of the *OnGetSQL* callback method, which uses the parameters supplied in the <altJSONSQLProvider>.

Class Member

```
Method GetSQL(ByRef pParm As %String,
 ByRef pSQL As %String,
 pCriteria As %ZEN.proxyObject,
 ByRef pPagingInfo As %String) As %Status
{
    Set pSQL = "SELECT ID,Name,Age FROM Sample.Person where Name %STARTSWITH ? or Age > ?"
    Quit $$$OK
}
```

The next example uses a class query to provide the SQL statement:

XML

```
<page xmlns="http://www.intersystems.com/zen">
  <altJSONSQLProvider id="jsonSQL"
    queryClass="Sample.Person" queryName="ByName" >
    <parameter paramName="1" value="Z"/>
  </altJSONSQLProvider>
  <dataGrid id="grid" controllerId="jsonSQL"/>
</page>
```

See the section "Referencing a Class Query" in the book *Using Zen Components* for more information on using a class query to supply an SQL statement.

The <altJSONSQLProvider> component defines the following attributes:

Attribute	Description
arrayName	The name of the output array. The default value is "children".
currPage	If the provider is using server-side data paging, this is the 1-based number of the current page.
OnGetSQL	This specifies a callback method that returns an SQL query (string) that drives this provider. This is identical in behavior to (and replaces) the sql property. The method can make it easier to create queries based on parameter values or search criteria passed via the criteria property.
pageSize	If the provider is using server-side data paging, this is the number of records in each page.
recordCount	If the provider is using server-side data paging, this is the total number of records.

The methods **%WriteJSONFromSQL** and **%WriteJSONStreamFromSQL** also support use of class queries and parameters. The first example shows use of a class query with **%WriteJSONFromSQL**.

```
set pid="Z"
set provider=##class(%ZEN.Auxiliary.altJSONSQLProvider).%New()
set provider.queryClass="Sample.Person"
set provider.queryName="ByName"
set param=##class(%ZEN.Auxiliary.parameter).%New()
set param.value=pid
do provider.parameters.SetAt(param,1)
do provider.%WriteJSONFromSQL("","",,1000,,provider)
quit
```

The next example provides two parameters to the SQL statement supplied to **%WriteJSONFromSQL**.

```
set pid="Z"
set provider=##class(%ZEN.Auxiliary.altJSONSQLProvider).%New()
set query="Select * from Sample.Person where Name %STARTSWITH ? or Age > ?"
set provider.sql=query
set param=##class(%ZEN.Auxiliary.parameter).%New()
set param.value=pid
set param1=##class(%ZEN.Auxiliary.parameter).%New()
set param1.value=20
do provider.parameters.SetAt(param,1)
do provider.parameters.SetAt(param1,2)
do provider.%WriteJSONFromSQL("","",,1000,,provider)
quit
```

The final example provides two parameters to the SQL statement supplied to **%WriteJSONStreamFromSQL**.

```
set pid="Z"
set provider=##class(%ZEN.Auxiliary.altJSONSQLProvider).%New()
set query="Select * from Sample.Person where Name %STARTSWITH ? or Age > ?"
set provider.sql=query
set param=##class(%ZEN.Auxiliary.parameter).%New()
set param.value=pid
set param1=##class(%ZEN.Auxiliary.parameter).%New()
set param1.value=30
do provider.parameters.SetAt(param,1)
do provider.parameters.SetAt(param1,2)
do provider.%WriteJSONStreamFromSQL(.stream,"",,1000,,provider)
quit
```

2.12 Zen Page Event Handling

Many components define event handlers such as *onclick*, *onchange*, etc. See the description of any Zen control component, such as <button>, <image>, or <submit>, in the "Zen Controls" chapter of *Using Zen Components*. Also see the description of zenEvent in the section "Client Side Functions, Variables, and Objects."

While a page is being displayed, Zen traps all events until the page finished building. This is to prevent the user from clicking on an element before the browser has finished building the `zenPage` object on the client side.

2.13 Zen Page URI Parameters

Sometimes it is useful to pass values to a Zen page via URI parameters. The ZENURL datatype parameter enables this feature.

Suppose that when you define a property in a Zen page class you apply the datatype parameter ZENURL to it, as follows:

Class Member

```
Property employeeID As %ZEN.Datatype.string(ZENURL="ID");
```

The previous example assigns a URI parameter with the name *ID* to the class property with the name employeeID. When this Zen page is requested by passing a URI to a browser, the value specified for *ID* is assigned to employeeID. The following URI:

```
MyApp.MyPage.cls?ID=48
```

Causes the following code to run:

```
 Set %page.employeeID = $GET(%request.Data("ID",1))
```

If the URI parameter assigned to a property does not pass the property's validation test for any reason, such as a value greater than the property's MAXVAL, the page is not displayed and an error message is displayed instead.

2.14 Zen Layout Handlers

The actual work of laying out a group component's children is handled by the %ZEN.LayoutManager class. It is possible to implement layout strategies in addition to those normally provided. If you wish to serve the page using your own, custom layout handler code, you can.

Zen provides a simple, built-in layout strategy with the intention of making rapid application development easier. Components start at top left of the page and work their way to the right and bottom. You can achieve a lot, very quickly, by manipulating nests of vertical and horizontal groups.

There may be cases when your layout needs finer control. If so, ultimately it might suit your project best to develop your own layout handler for use with Zen. For these cases, Zen provides an **onlayoutHandler** callback method. This client-side method, if defined, is called when a page is first displayed and whenever the size of the page is changed thereafter. This provides a way to define code that adjusts the size and position of components whenever the size of the containing page is changed.

For example:

Class Member

```
ClientMethod onlayoutHandler(load) [ Language = javascript ]
{
   // adjust size of lookout menu
   var menu = zen('lookout');

   // find div for titleBox & mainMenu
   var title = zen('titleBox');
   var divTitle = title.getEnclosingDiv();

   // find height of window
   var winHeight = zenGetWindowHeight();

   // adjust size of menu
   var sz = winHeight - (parseInt(divTitle.offsetHeight)) - 20;
   menu.setSize(null,sz);
}
```

2.15 Zen Utility Methods

The %ZEN.Utils class includes a number of utility methods for performing various common functions. Most of these classes are for internal use by the Zen library.

ZENTest.LogPage in the SAMPLES namespace is an example of a page class that uses some of these utility methods to respond to the user's selection of whether or not to enable logging for the application. The page class provides:

- A <tablePane> definition for the event log display, plus the following <checkbox>:

  XML

  ```
  <checkbox id="cbEnabled" caption="Logging Enabled"
            onchange="zenPage.enabledChange(zenThis);"/>
  ```

- This client-side method:

  Class Member

  ```
  ClientMethod enabledChange(cb) [ Language = javascript ]
  {
    this.EnableLog(cb.getValue()==1 ? true : false);
    this.refreshLog();
  }
  ```

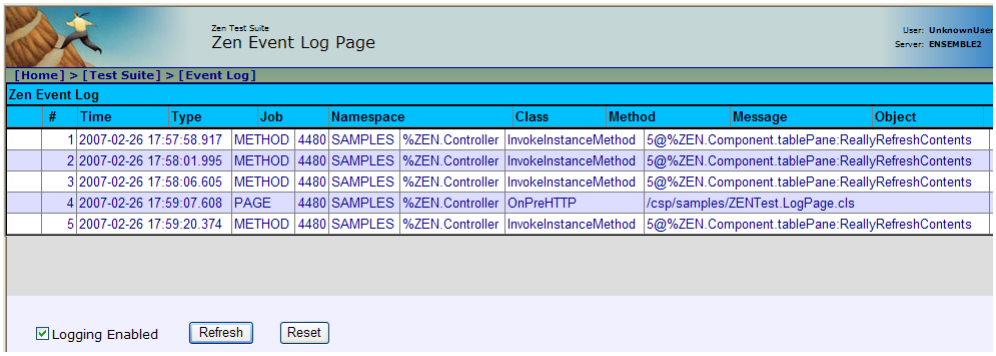
- And this Zen method:

  Class Member

  ```
  ClassMethod EnableLog(flag As %Boolean = "") As %Boolean [ ZenMethod ]
  {
    If (flag = "") {
      Set flag = ##class(%ZEN.Utils).%LoggingEnabled()
    }
    If (flag) {
      Do ##class(%ZEN.Utils).%StartLog()
    }
    Else {
      Do ##class(%ZEN.Utils).%StopLog()
    }
    Quit 1
  }
  ```

The result is as follows:

Figure 2–1: Sample Log of Zen Events



To read more about %ZEN.Utils, start the InterSystems online documentation system and choose **Class Reference Information** for the %SYS namespace.

3

Zen Security

You can provide application-level security for Zen using a combination of a Zen form and system configuration settings. For background information, see the section "Application Resources and Their Privileges" in the "Assets and Resources" chapter of the *Caché Security Administration Guide*. For instructions, see the next section, "Controlling Access to Applications."

There are additional techniques for controlling access to individual pages and components within a Zen application. See the sections later in this chapter, "Controlling Access to Pages" and "Controlling Access to Components."

3.1 Controlling Access to Applications

By default, when a user starts your Zen application it prompts the user to log in by displaying the standard Caché username and password dialog box. If you wish your application to present a custom login form, you must create and configure this form as follows:

1. Create a new Zen page class.
2. Within XData Contents, provide a <loginForm> component. <loginForm> is a special-purpose component that simplifies the development of login pages and ensures that login processing is handled entirely by CSP.

   Important: To create a custom login form, you *must* use <loginForm>. Other approaches for creating login pages in Zen applications can cause problems of various kinds.

3. Make sure that the <loginForm> contains the following controls:
   - `<text name="CacheUserName"/>` for the username
   - `<password name="CachePassword"/>` for the password

   Ensure that you provide a *name* attribute for each of these controls and that the corresponding *name* values are exactly as shown above: `"CacheUserName"` and `"CachePassword"`. You may provide other attributes for the controls as desired; a *label* is useful for each control so that the user knows what to enter.
4. Provide a <submit> button. By default, pressing Enter triggers a form submit. If you do not want this behavior, add the attribute `ondefault="return true;"` to the <loginForm> definition. Your minimal user login form now looks like this:

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">
  <loginForm id="loginForm" >
    <text name="CacheUserName" label="User:" />
    <password name="CachePassword" label="Password:" />
    <submit caption="Login" />
  </loginForm>
</page>
```

5. Define other characteristics of the form as desired.
6. Start the Management Portal.
7. Navigate to the **Web Applications** page (**System Administration** > **Security** > **Applications** > **Web Applications**).

   Note: For background information about this step, see the "Zen Application Configuration" section in the chapter "Zen Application Programming."

8. Find the appropriate web application in the list and click its **Edit** button. The **Web Applications** page (**System Administration** > **Security** > **Applications** > **Web Applications**) displays.
9. In the **Login Page** field, enter the package and class name of your new Zen page class. Be sure to include the **Web application Name** at the start of the path, for example:

   `/csp/myNamespace/myPackage.myLoginPage.cls`

10. Click **Save**.

A change password form is also possible using the <password> component. You must use `"CacheOldPassword"` as the name of the control containing the old password value, `"CachePassword"` as the name of the control containing the new password value, and `"CacheRepeatPassword"` as the name of the control where the user retypes the new password value for verification. Also, when you configure the application you must identify this Zen page as the **Change Password Page** rather than the **Login Page**:

XML

```
<page xmlns="http://www.intersystems.com/zen" title="">
  <form>
    <text name="CacheUserName" label="Name:"/>
    <password name="CacheOldPassword" label="Old Password:"/>
    <password name="CachePassword" label="New Password:"/>
    <password name="CacheRepeatPassword" label="Retype New Password:"/>
    <submit caption="Submit"/>
  </form>
</page>
```

3.2 Controlling Access to Pages

Each Zen page has a class parameter called RESOURCE that, if defined, provides the name of a system Resource for which the current user must hold USE privileges in order to view this page or to invoke any of its server-side methods from the client.

For background information about class parameters such as RESOURCE, see the "Page Class Parameters" section in the chapter "Zen Application Programming." To understand what a system Resource is and what the USE privilege means, see the section "Application Resources and Their Privileges" in the "Assets and Resources" chapter of the *Caché Security Administration Guide*.

3.3 Controlling Access to Components

Each Zen component (subclass of %ZEN.Component.component) has a server-only property called %resource that determines whether or not this component should be added to the set of page components. The component projects this property to XML as an attribute called *resource*. You can use this resource when adding a component to XData Contents. For example:

XML

```
<button id="myButton" caption="Press Me" resource="ADMIN" />
```

If a *resource* value is specified, the current user must hold USE privileges on this resource or the component is not added to the set of page components when the user attempts to display the page. This property is not available from the client.

3.4 Legacy Application Access

It is possible to use Web Application Configuration page of the Management Portal, as described in "Zen Application Configuration," to permit "Unauthenticated Access." This setting allowed applications implemented prior to Cache 5.2 to continue working without requiring changes. InterSystems does not recommend using this setting on production systems for *any* application. The only time this setting should be used on a production server is for a legacy application for a short period of time until the existing application is modified to run securely with access controlled by Roles and Resources. There may be cases on development systems where it is convenient to allow an application to run with "Unauthenticated Access" until the application under development is ready to interact with Caché Security. But you must be aware that %UnknownUser of such a web application is running with %ALL as a role and likely has the privileges of the account used to start and stop the system with regard to operating system level privileges.

4

Zen Localization

When you *localize* the text for an application, you create an inventory of text strings in one language, then establish a convention for substituting translated versions of these messages when the application locale is different. Most of the information you need to localize Zen pages can be found in the "Localizing Text in a CSP Application" chapter in the book *Using Caché Server Pages (CSP)*. Everything described in that chapter for development of CSP pages also applies to Zen pages.

There are a few additional details that apply only to Zen. This chapter:

- Summarizes the CSP information
- Describes Zen extensions to CSP techniques

Important: In order for Zen localization techniques to work in any Zen class, the class must provide a value for the DOMAIN class parameter.

4.1 CSP Localization

Important: This topic briefly outlines material from the "Localizing Text in a CSP Application" chapter in the book *Using Caché Server Pages (CSP)*. If you are new to CSP, please read the other chapter before continuing in this book.

For a simple demonstration of a localized web application, enter the following URI while Caché is running. Substitute your Caché web server port number for 57772:

```
http://localhost:57772/csp/samples/language.csp
```

4.1.1 Localization Practices

Caché supports the following process for localizing web application text:

1. Developers determine where text strings are displayed in the application user interface.
2. Developers create an XML message file that contains text strings in the original language.
3. Developers import the XML into a Caché namespace.

   This adds new entries to the message dictionary in that namespace.
4. Developers give the XML to a team of translators.

5. Translators produce a new XML message file by substituting translated text for the original.
6. Developers import the new XML into a Caché namespace.

   Translated and original texts coexist in the message dictionary.
7. At runtime, the application chooses which text to display based on the browser default language.

4.1.2 message dictionary

A message dictionary is a simple database of text strings organized by domain name, language name, and message ID:

- The *text* of each message is a string of up to 32K characters. A message may consist solely of text, or it may also contain one or more parameters specified by `%1`, `%2`, etc.
- A *domain* name is any arbitrary string. It identifies a group of related text items, such as all messages for a specific application or page.
- A *language* name is an all-lowercase language tag that conforms to RFC1766. It consists of one or more parts: a primary language tag (such as `en` or `ja`) optionally followed by a hyphen (-) and a secondary language tag (`en-gb` or `ja-jp`).
- A *message ID* is any arbitrary string; it uniquely identifies a message. The message ID only needs to be unique within a domain. You may assign a message ID or allow Caché to assign one, depending on the conventions you use to create the message.

Each user-defined namespace in Caché stores its message dictionary in a subscripted global called *^CacheMsg*. The order of subscripts in *^CacheMsg* is domain, language, and message ID.

4.1.3 $$$Text Macros

Important: For $$$Text macros to work in any Zen class, the class must provide a value for the DOMAIN class parameter.

The easiest way to create a message dictionary is to automatically generate message dictionary entries at compile time, by seeding the CSP class code with $$$Text macro calls. $$$Text automatically creates the message at compile time and generates the code that retrieves the message at runtime, as this topic explains.

When it comes time to translate the application messages into another language, you can export the full list of messages for the original language out of the message dictionary by running an Export command. This generates a complete XML message file in the original language. See the article *String Localization and Message Dictionaries*.

The return value of a $$$Text macro is a %String. The correct $$$Text macro to use depends on the format you need for this output string:

- $$$Text
- $$$TextJS (applies JavaScript escaping to the $$$Text result)
- $$$TextHTML (applies HTML escaping to the $$$Text result)

The `%String` returned by $$$Text may be assigned to a variable, which you can use to represent the message in subsequent calls. For example:

ObjectScript

```
Set tmsg = $$$TextJS("Error saving production")
&js<alert('#(tmsg)#: #($ZCVT($ZE,"O","JS"))#');>
```

Or, you can simply insert a $$$Text macro anywhere you need a string:

ObjectScript

```
&js<alert('#($$$TextJS("Error saving production"))#: #($ZCVT($ZE,"O","JS"))#');>
```

4.1.3.1 $$$Text Arguments

$$$Text has the arguments *text*, *domain*, and *language* as described in the following table. Only the first argument, *text*, is required.

Argument	Description
text	Non-empty string. *text* must be a literal string. It cannot use any type of expression syntax. The format used for *text* may be: "*actualText*" Or: "*@textId@actualText*" Where *textId* is a message ID and *actualText* is the text of the message. The string *actualText* may consist of any of the following items, separately or in combination: • Simple text, as permitted by the file format • Substitution arguments `%1`, `%2`, `%3`, or `%4` • HTML formatting • A string expression in ObjectScript format If provided, the *textId* is used as the message ID. If *@textId@* is not specified, the system generates a new *textId* by calculating the 32–bit CRC (Cyclic Redundancy Check) of this text. If the *textId* is specified and a message already exists with this ID, the existing message is checked to see if it has the same text as *actualText*. If not, an error is reported.
domain	(Optional) A string specifying the domain for the new message. If not specified, *domain* defaults to the value of the DOMAIN class parameter at compile time and `%response.Domain` at runtime.

Argument	Description
language	(Optional) An RFC1766 code specifying the language. Caché converts this string to all-lowercase. If not specified, *language* defaults as follows: • At compile time: If there are no previous compilations in this domain, the default is the general default language for the system. This value can be found in the variable `$$$SessionLanguage`. If there was a previous compilation in this domain, the language specified during that compilation is the default. • At runtime: The default is `%response.Language`, or if no value is defined for `%response.Language`, the default is `%session.Language` which contains the same value as `$$$SessionLanguage`. Tag-based CSP pages automatically acquire a value for `%response.Language` from browser settings, so it is available as a default language and also as a value for `%session.Language`. This is not true for class-based CSP pages, which must explicitly set a value for `%response.Language` to use it as a default. Setting a value `%response.Language` also has the side effect of setting `%session.Language` to the same value. You can assign a value to `%response.Language` by giving it the return value of the %Library.MessageDictionary class method **MatchLanguage**. Given a list of languages and a domain name, this method uses HTTP 1.1 matching rules (RFC2616) to find the best-match language within the domain.

4.1.3.2 $$$Text at Compile Time

When you compile a class that contains calls to $$$Text, $$$TextJS, or $$$TextHTML macros, each call generates a message in the message dictionary, with *text*, message ID, *domain*, and *language* as provided by the macro arguments.

4.1.3.3 $$$Text at Runtime

If the message text contains arguments (`%1`, `%2`, `%3`, `%4`) you must specify the corresponding substitution text before displaying the text. Since $$$Text returns a string, you can use any string operation native to your coding language. For example, in JavaScript:

```
var prompt = '#($$$TextJS("Remove user %1 from this Role?"))#';
prompt = prompt.replace(/%1/g,member.userName);
```

You can also use the $$$Text string as the first argument of the **%response.FormatText** method or a $$$FormatText macro. For details, "Other Options for Displaying Localized Strings" in the chapter "Localizing Text in a CSP Application" in *Using Caché Server Pages (CSP)*.

4.2 Zen Localization

Important: For the following conventions to work in any Zen class, the class must provide a value for the DOMAIN class parameter.

Zen pages are CSP pages. All CSP localization practices also apply to Zen pages. However, Zen adds the following conventions to make localization even easier:

- Any Zen property that has its ZENLOCALIZE parameter set to 1 (true) automatically generates a message dictionary entry. Caché generates the message dictionary using similar conventions as when you use `textid=""` in <csp:text> and other localizable CSP tags. That is:
  - The message text is the string value of the property
  - The message ID is the 32–bit CRC of the text
  - The domain takes the value of the DOMAIN class parameter
  - The language takes the value of the browser default language (at runtime)
- The Zen datatype class, %ZEN.Datatype.caption, has a default value of 1 (true) for its ZENLOCALIZE parameter, so any property defined as having a type of %ZEN.Datatype.caption is automatically localized as described above.

4.2.1 Localization for Zen Components

If a Zen component has any string-valued properties with its datatype parameter ZENLOCALIZE set to 1, then when Zen generates the **%CreatePage** method from the <page> description, Zen automatically localizes the text supplied for those properties.

It is convenient to give all string-valued properties the Zen datatype %ZEN.Datatype.caption, which has ZENLOCALIZE automatically set to 1, but you can also set ZENLOCALIZE directly by including the datatype parameter `ZENLOCALIZE=1` in the property definition.

If *all* of the following conditions are true:

- You are using components that have properties with ZENLOCALIZE set to 1
- You place these components within the <page> element in XData Contents
- Your Zen page class has a DOMAIN parameter value defined

Then the code that Zen generates for the **%CreatePage** method automatically calls $$$Text to localize each of these properties. Of course, you must provide the translations yourself, as described in "Localization Practices." However, if you do this and also use ZENLOCALIZE and $$$Text as described in this section, all of the mechanisms for delivering these translated strings work automatically. The result is that Zen automatically serves the correct translation to the user by detecting the language locale for the browser or the Caché server at runtime.

The following is a simple example:

```
<button caption="OK" />
```

Generates the following code in **%CreatePage**:

```
Set bttn.caption = $$$Text("OK")
```

Important: If you programmatically build pages (without using <page>) there is no automatic localization of component properties, even if `ZENLOCALIZE=1`. In this case your code must issue a call to $$$Text for each one of these strings, as shown in the previous example of generated code for a button caption.

4.2.2 Localization for Custom Components

If you are creating a new component, you can define properties that should be localized as datatype %ZEN.Datatype.caption:

```
Property title As %ZEN.Datatype.caption;
```

Within your component, to refer to the property simply use the property name with double dot syntax (`..title` in this example). You do not need $$$Text within your custom component code unless you have some static text that you wish

to localize. Whenever a <page> references your component, your property is automatically localized in the same way as built-in Zen components. For example:

```
<MyComponent title="AAA" />
```

Generates code like this in **%CreatePage**:

```
Set MyCmp = $$$Text("AAA")
```

4.2.3 Localization for Client-side Text

Zen supports a variant of $$$Text that you can use within JavaScript methods. This feature greatly simplifies the use of localized strings in client-side code.

The syntax for JavaScript localization is similar to the $$$Text macro in ObjectScript:

```
var str = $$$Text('This is a localized string.');
```

When a method containing this code runs in a Zen page, $$$Text returns a localized string if localization has been enabled by setting the DOMAIN parameter, and a localized version of the string is available. $$$Text takes an optional second argument, which is a DOMAIN name. This argument lets you override the default localization domain:

```
var str = $$$Text('This is a localized string.', 'MyDomain');
```

JavaScript does not support macros, so $$$Text is implemented as a function that returns a string. Strings in JavaScript can be quoted with either single quotes (') or double quotes ("), and both conventions are supported. Both arguments to $$$Text must be literal (quoted) strings or the function does not return localized values. The complete $$$Text() expression must be written on a single line of code. All occurrences of $$$Text('string') are processed, wherever they appear in the JavaScript, including in comments and strings. This does not effect the behavior of the code, but may add extra entries to the string table.

You can use explicit id values for strings, such as "@33@MyString", in the same manner as the server-side $$$Text.

In Caché version 2010.2, the $$$Text function exists but simply returns the string it is passed.

The JavaScript function, $$$FormatText is also supported. This substitutes all occurrences of %1,%2,%3,%4 with the values of up to four parameters:

```
alert($$$FormatText('Ok to delete file %1?', filename));
```

$$$Text automatically captures string content and adds it to the localization global. The macro preprocessor does this in the case of the $$$Text macro in ObjectScript. As there are no macros in JavaScript, the compiler that converts JavaScript methods into client-side functions scans JavaScript for occurrences of $$$Text('string','domain') and pulls the first argument (and the second if it is present) and constructs a JavaScript array of localized values, which is served as part of the page. The $$$Text function uses this array to convert strings to the localized version. The generation of the array uses the server-side $$$Text macro, which ensures that the JavaScript localized strings are created using the same mechanism as server-side strings.

When a Zen page is rendered, the array of localized strings is created for every type of component and superclasses of that component found on the page. If a component is added to a page dynamically and Zen determines that the class definition for this component must be dynamically loaded, then localized strings are generated for this component as well.

4.2.4 Localization with zenText

An additional client side equivalent of $$$Text is the `zenText` JavaScript function:

`zenText(`*id*`,`*p1*`,`*p2*`,`*p3*`,`*p4*`)`

This function finds a localized text message from a set of client-side text resources, where:

- *id* is the id of the text message.
- *p** variables are optional. Zen substitutes them for %1, %2, and so on if these variables appear in the message text.

If no localized strings are defined for the supplied *id*, `zenText` returns a default string.

You can define a set of localized text messages by overriding the **%OnGetJSResources** callback method in your Zen page class, and filling it with `Set` statements, such as those that define the resources `MyMessage` and `FieldValidation` in the following example:

Class Member

```
Method %OnGetJSResources(ByRef pResources As %String) As %Status [ Private ]
{
  Set pResources("MyMessage") = $$$Text("This is my message")
  Set pResources("FieldValidation") = $$$Text("Field %1 has an error")
  Quit $$$OK
}
```

The **%OnGetJSResources** method fills in an array containing pairs of resource identifiers and text strings. **%OnGetJSResources** is invoked automatically, allowing you to access these resources from JavaScript using the `zenText` function whenever you need it; for example:

```
alert(zenText('MyMessage'));
alert(zenText('FieldValidation','SSN'));
```

Note: It is possible to define client side resources that apply to every page in the application. To do this, override the **%OnGetJSResources** method within the application class rather than in the page class.

As with $$$Text, you must provide the translations for strings such as `"This is my message"` yourself, as described in "Localization Practices." However, if you do this and also use `zenText` and **%OnGetJSResources** as described in this section, all of the mechanisms for delivering these translated strings work automatically. The result is that Zen automatically serves the correct translation to the user by detecting the language locale for the browser or the Caché server at runtime.

There are some localized text strings already defined as resources for client side localization in all Zen page classes. The following display lists them. The syntax in this example is not complete, because the right-hand side of the display is truncated to keep the lines short. However, each line shows enough to indicate what to expect from this resource if you invoke it using `zenText`:

```
Set pResources("zenMonthNames") = $$$Text("January,February,March,April,May,June
Set pResources("zenMonthShortNames") = $$$Text("Jan,Feb,Mar,Apr,May,Jun,Jul,Augu
Set pResources("zenDayNames") = $$$Text("Sunday,Monday,Tuesday,Wednesday,Thursda
Set pResources("zenDayShortNames") = $$$Text("S,M,T,W,T,F,S","%ZEN")
```

Note: For information about JavaScript functions other than `zenText`, see "Client-Side Functions and Variables" in the "Zen Page Classes" section of the chapter "Zen Pages."

5

Custom Components

One of the most powerful features of the Zen framework is that it lets you easily develop new, reusable components that automatically work with other Zen components.

From its first chapter, "Introducing Zen," the book *Using Zen* asserts that the Zen application development framework is extensible. The claim is certainly true. However, Zen anticipates many of the features that you might consider adding yourself. It is worthwhile to examine all of the available Zen components before beginning a custom development effort. Possibly, the features you want are available in a Zen component that already exists.

The book *Using Zen Components* presents the details of existing Zen components by category: tables, meters, charts, forms, controls, menus, popups, and other. After examining all of these chapters, you might find that you still wish to create custom components. If so, continue with this chapter to view the options and instructions for doing so.

Important: InterSystems strongly recommends that you do not create any classes in a package called ZEN.Component using any combination of uppercase and lowercase characters. Creating a package called ZEN.Component breaks the Zen framework for generating client-side code.

In general, the most common way to extend Zen is to create a custom component. Custom components fit into the existing Zen framework with minimal effort. They provide the style and behavior that you need while still obeying the Zen layout conventions and leveraging the Zen client/server framework. The following list outlines the options for creating custom components, and provides links to the sections in this chapter that describe each option:

- "Composite Components" describes how to build a composite component. A composite arranges a group of existing, built-in Zen components in a particular way. You can place this arrangement on the page as if it were a single component.
- "Overriding Component Style" explains how to make simple changes to the style of a built-in Zen component by subclassing it and overriding its XData Style block.
- "Creating Custom Components" describes how to extend the roster of Zen components by writing a new component class that is a subclass of one of the standard base classes for Zen components. The resulting class is automatically projected as XML for use in any Zen page XData Contents block.

  This chapter provides several sections that support "Creating Custom Components":

  – "Custom Style" places an XData Style block in the subclass.

  – "Custom Properties" defines properties in addition to those inherited from the base class.

  – "The %DrawHTML Method" is a required method that provides the HTML (or SVG) needed to create the visual representation on the client side.

  – "Custom Methods" defines methods in addition to **%DrawHTML** and those inherited from the base class.

  – "Sample Code" provides a sample custom component class and shows how to reference it from the XData Contents block in a Zen page class.

- "Creating Custom Meters" shows how to add a new type of meter to Zen by subclassing the base meter class and adding the method calls required to render the new meter's SVG contents on the client.

5.1 Composite Components

A composite component assembles a group of built-in Zen components and lets you refer to the group from XData Contents as if it were a single component. Zen places the children of the composite within the page's object model as if XData Contents referenced them explicitly. This shorthand is convenient, and it also prevents errors when you want to repeat identical component arrangements on different pages.

The following table is your checklist for building a composite component class. Not every item in the checklist is necessary for every composite. This checklist lists every possible item, and notes those that are optional. See the example of a composite component class that follows the table.

Table 5–1: Extending Zen with Composite Components

Required	Optional	Description
Subclass		Extend %ZEN.Component.composite to create a new class. The example creates a composite component class called myComposite in the MyApp package. Any class that extends %ZEN.Component.composite is a group, with all of the attributes described in the section "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
Parameters		Provide class parameters and appropriate values. For example, the NAMESPACE parameter is required to assign a namespace to the composite. You must reference this namespace when you place the composite component on a Zen page.
	Properties	Provide properties, if needed. The example does not require any new properties for the subclass.
	Methods	Provide supporting methods as needed for component behavior. The example provides an event handler method that is activated when the user clicks one of the buttons in the composite.
XData Contents		Within the composite component class, define an XData Contents block that uses <composite> as its top-level element. The example provides two <button> elements within <composite>.
XML Element		To add a composite element to a page, place its corresponding XML element in the page class XData Contents block. There is a specific syntax for this; see the example following this table.
Compile Order		For a discussion of compile order that applies to composites as well as other types of custom component, see the section "Compile Order for Custom Component Classes."

The following is a sample composite component class that provides two buttons:

```
Class MyApp.myComposite Extends %ZEN.Component.composite
{

  /// Define xml namespace for this component
  Parameter NAMESPACE = "http://www.intersystems.com/myApp";

  /// Contents of this composite component:
  XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
```

```
  {
    <composite>
      <button caption="OK"
              onclick="zenThis.composite.okBtn();"/>
      <button caption="Cancel" />
    </composite>
  }

  ClientMethod okBtn() [Language = JavaScript]
  {
    alert('ok');
  }
}
```

The correct way to reference this composite component from an XData Contents block is as follows:

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
<page xmlns="http://www.intersystems.com/zen"
      xmlns:myApp="http://www.intersystems.com/myApp">
  <myApp:myComposite />
</page>
}
```

Where:

- The <page> element references the namespace defined for the composite component, and defines an alias for it, by providing the following attribute within the opening <page> element:

  `xmlns:myApp="http://www.intersystems.com/myApp"`

  This statement defines an alias `myApp` for the namespace that was defined in the composite component class

  `Parameter NAMESPACE = "http://www.intersystems.com/myApp";`

- The <myApp:myComposite> element references the myComposite component. This reference has the form:

  <*alias*:*component*>

  where *alias* is the namespace alias defined in the <page> element (`myApp` in this case) and *component* is the name of the custom component class (`myComposite` in this case).

5.1.1 The composite Property

When you work with composite components programmatically, every component within a composite has a property called composite that points to the containing composite object. This makes it easy to define actions for a composite that refer to methods of the composite class. You can see this in the first <button> element from the example above:

XML

```
<button caption="OK"
        onclick="zenThis.composite.okBtn();"/>
```

The *onclick* attribute value for this <button> definition uses *zenThis.composite* to represent the containing composite object. This convention allows the <button> to call the **okBtn** method that is defined within the composite class.

5.1.2 Composites and Panes

To review panes and how their contents are defined, see the section "Panes" in the "Zen Layout" chapter of *Using Zen*.

When a <pane> component is placed within a composite element, the contents of the pane can be defined within the composite class (or subclass of the composite). At runtime, the composite first looks for a pane with a given name within the current page; if it does not find it there, it looks within the composite class.

5.2 Overriding Component Style

If you have subclassed one of the built-in Zen components, you can override the styles defined for that component. See "Overriding Built-in Styles" in the "Zen Style" chapter of *Using Zen*. Briefly noted, the instructions are:

1. Determine the relevant CSS style name
2. Subclass the built-in Zen component
3. Use the Zen Style Wizard to edit XData Style.

However, if you want to define an entirely new CSS style (with a new name) and apply that style to a component, there are additional steps to perform. You must not only define the CSS style, but also reference it from the class code that renders the custom component as HTML. At this point, you are truly creating a custom component. For details, see the section "Creating Custom Components."

5.3 Creating Custom Components

Each Zen component is implemented as a class and is completely self-contained. The component class specifies its own behavior (server and client methods) and appearance (SVG or HTML, DHTML, and stylesheets) in one logic unit: the class. Each component class has an XML projection. This projection consists of:

- An XML element with the same name as the class (<page>, <html>, <hgroup>, etc.)
- XML attributes, which are properties of that class (width, height, etc.)

To build a Zen page you place a selection of the available XML elements and attributes in a well-formed XML document in the Zen page class. The container for this document is called XData Contents. At compile time, Zen generates the code required to display the page in the browser with the layout, style, and behavior that you have chosen for its components. At runtime, Zen handles all of the display details and correctly executes any snippets of JavaScript, HTML, or CSS that are embedded or referenced in the component or page classes.

Zen components provide all of these features automatically because they inherit them from their base classes. This is something you can do yourself. To create a custom component, simply choose the appropriate base class and extend it. Your new class automatically gets the following features:

- An XML projection to use in the page class XData Contents
- Any properties, methods, parameters, or other characteristics of its parent(s)
- Appropriate handling of any client-side material in the class:
  - JavaScript
  - CSS style definitions
  - HTML
  - SVG
- Properties can be exposed as settings, to be observed and modified using the client-side **getProperty** and **setProperty** methods.

The following table is your checklist for creating a custom component class. To understand syntax and usage details about each item in the table, use the links within the table, or consult the topics following the table.

Table 5–2: Extending Zen with Custom Components

Required Task	Optional Task	Description
	Package	InterSystems suggests you use one class package for all your custom component classes. This assists with various issues including compile order.
XML Namespace		InterSystems suggests you use one XML namespace for all your custom components.
Subclass		Choose a base class from a short list of candidates.
	Parameters	Provide class parameters, such as NAMESPACE or INCLUDEFILES.
	XData Style	Within the component class, provide an XData Style block that defines CSS styles.
	Properties	Provide properties, if needed. The various properties of a component can be exposed as settings, and can be observed and modified using the client-side **getProperty** and **setProperty** methods.
%DrawHTML		Visual components must draw the HTML (or SVG) needed to create their client-side visual representation. You must ensure that the **%DrawHTML** method in the custom component class references any new CSS styles defined in the class.
	Methods	In addition to **%DrawHTML**, create or override component methods as needed to ensure the correct component behavior.
Compile Order		You might encounter compile order issues when a Zen page references a custom component, or when a Zen page references a composite component that references a custom component.
XML Element		To add a custom component to a page, place its corresponding XML element in the page class XData Contents block. There is a specific syntax for doing this; see the section "Sample Code."

5.3.1 Package for Custom Component Classes

Important: InterSystems strongly recommends that you do not create any classes in a package called ZEN.Component using any combination of uppercase and lowercase characters. Creating a package called ZEN.Component breaks the Zen framework for generating client-side code.

InterSystems suggests you use one class package for all your custom component classes. There are several reasons for these conventions:

- The Zen framework automatically generates JavaScript and CSS stylesheet include files for your custom components. These files are generated on a per-class-package basis, so it is much easier to organize this if your custom components are contained within their own package.
- You might encounter compile order issues when a Zen page references a custom component, or when a Zen page references a composite component that includes a custom component. There are several ways to prevent compile order issues, as described in the next section, "Compile Order for Custom Component Classes." One approach is to place all custom components in one package and ensure that your build procedure compiles that package before any package that contains Zen page classes.

When you reference the custom component in a Zen page XData block, do not use the package name. Use only the class name, as shown in the example in the next section, "XML Namespace for Custom Component Classes."

5.3.2 XML Namespace for Custom Component Classes

Custom components need their own XML namespace to avoid naming conflicts with other components. Define and reference this namespace as follows:

- Provide the NAMESPACE parameter in each custom component class that you create. In the following example:
  - The custom component class name is:

    `myCustom`

  - The namespace is:

    `http://www.mycompany.com/mycomponents`

Class Definition

```
Class MyPackage.myCustom Extends %ZEN.Component.textarea
{

/// This is the XML namespace for this component.
Parameter NAMESPACE = "http://www.mycompany.com/mycomponents";

/// This Style block contains component-specific CSS style definitions.
XData Style
{
}

/* Here is the rest of the class... */
}
```

- Define a prefix for the namespace at the beginning of each XData block that references a custom component. Then, when referencing the custom component, use the prefix. In the following example:
  - The custom component class name is:

    `myCustom`

  - The namespace is:

    `http://www.mycompany.com/mycomponents`

  - The prefix is:

    `myco`

Class Member

```
XData demoPane [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<pane xmlns="http://www.intersystems.com/zen"
      xmlns:myco="http://www.mycompany.com/mycomponents"
      valign="top" >
  <myco:myCustom id="DemoText" name="DemoText"
                 label="Operator Instructions:"
                 cols="32" rows="8"
                 title="Type comments and drag text into the editing area"
                 hint="Describe how to use the product at this step."
                 enclosingClass="myValid" />
  <hgroup align="center">
    <button caption="Clear" onclick="zenPage.clearAllValues()" />
    <button caption="Revert" onclick="zenPage.revertToSaved()" />
    <submit caption="Save" />
  </hgroup>
</pane>
}
```

InterSystems suggests that you define one XML namespace to use for all of your custom components, and use that XML namespace only for custom components.

5.3.3 Compile Order for Custom Component Classes

You might encounter compile order issues when a Zen page references a composite or custom component. The symptom of a compile order issue is when you receive a compile-time error message that you can fix by simply recompiling your code. In these cases, to prevent error messages and compile the application correctly, your build procedure must compile the classes in order beginning at the lowest level class and working upward, as follows:

1. Custom components. These are the lowest level classes for the compile order, so compile them first.
2. Composite components. These might reference custom components, so compile them after any custom components.
3. Pages. These might reference composite or custom components, so compile pages last, after you have already compiled custom and composite components.

There are several ways to prevent issues with compile order in Zen applications:

- Allow all classes to remain in the same package, but provide the DependsOn keyword in higher-level classes that need to reference lower-level classes. This is easiest to achieve when there are a small number of classes and not much layering of references to other classes. Simply place a DependsOn statement at the beginning of each higher-level class definition that references a lower-level class. The DependsOn value must identify the package and class name of the lower-level class. For example:

  Class Definition

  ```
  Class MyPackage.MyZenPage Extends %ZEN.Component.page
                            [ DependsOn = MyPackage.myCustom ]
  {
  /* Here is the rest of the class... */
  }
  ```

- For larger applications, InterSystems recommends that you place all custom component classes in one package, all composite components in a second package, and all Zen page classes in a third package. Ensure that your build procedure compiles these three package in the proper order, from the lowest to the highest level, by invoking separate commands to compile each package individually.
- It is also possible to allow larger applications to keep all classes in the same package if you assign the classes to separate compile groups. You can do this by adding the System keyword in the lower-level classes to sort them into compile groups, as follows:

  1. Set custom component classes to `[System=3]` so these are compiled first.
  2. Set composite components to `[System=4]` so these are compiled next.
  3. Provide no System keyword or value for the remainder of the application classes, so these are compiled last.

  You can set the System keyword in Studio. Setting the value for this property to 3 or 4 has the desired result. A value of 0 is the default value; it is the same as having no System value set for the class.

  CAUTION: InterSystems strongly recommends you do not use the value 1 or 2 for the System keyword. Values larger than 4 are not supported and the Studio Inspector does not allow you to choose them.

5.3.4 Zen Component Wizard

Use the Studio New Zen Component Wizard to create a new Zen component class. You can start the wizard using the Studio **File New** command, selecting the **Zen** tab, and clicking the **New Zen Component** icon.

5.3.5 Component Base Classes

When creating a new component, you must choose a base class for it. There are five significant options to choose from. The following figure and table summarize the choices.

Figure 5–1: Base Classes for Custom Components

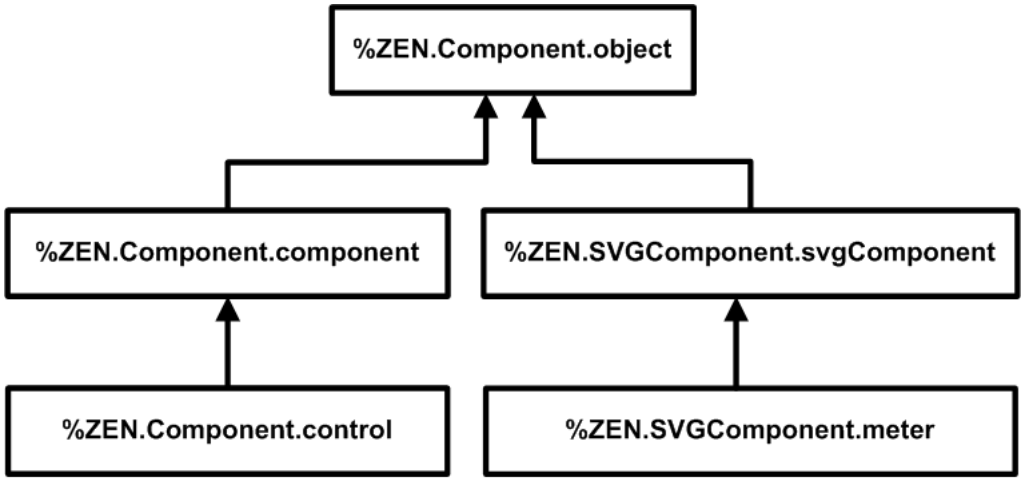


Table 5–3: Custom Component Base Classes

Base Class	Purpose	Inherited Attributes
%ZEN.Component.component	Visual, HTML based component.	Zen components
%ZEN.Component.control	HTML based component can be placed within a form for the user to enter a value.	Zen components, Zen controls
%ZEN.SVGComponent.svgComponent	Visual, SVG based component.	SVG components
%ZEN.SVGComponent.meter	Dynamically updated SVG graphic that displays a value.	SVG components, meter components
%ZEN.Component.object	Non-visual component. Typically this is a helper object that is used by visual components and requires a client-side object representation.	—

5.3.6 Component Class Parameters

Depending on the base class you choose for your custom component, various class parameters are available for your use. Each of the component base classes sets its class parameters to values that are typically useful for that type of component. You can reset these values if you wish. The following table describes the class parameters that are most likely to be of interest.

For others, you may examine the Class Reference documentation for the your base class, as follows: Start the InterSystems online documentation. Select **Class Reference** from the menu bar at the top of the documentation home page. Choose the %SYS namespace and %ZEN.Component or %ZEN.SVGComponent package. Click on the class name.

Table 5–4: Component Class Parameters

Parameter	Meaning
DEFAULTVISIBLE	True (1) or false (0). The %ZEN.Component.object base class sets DEFAULTVISIBLE to 0. Any Zen component that needs to be visible on the page overrides this value to 1. All of the other base classes listed in the previous table set this value to 1, so if your custom component class inherits from these classes your component is visible by default.
INCLUDEFILES	Comma-separated list of JavaScript (.js) or Cascading Style Sheet (.css) files to include when this component is used on a page. Only simple file names may be used; for the physical path to these files, see the "Zen Application Configuration" section in the chapter "Zen Application Programming."
NAMESPACE	The XML namespace used for components. See the section "Namespace for Custom Component Classes." The default NAMESPACE value is `"http://www.intersystems.com/zen"`
POSTCOMPILEACTIONS	Comma-separated list of actions to perform after this class is compiled. The list may be empty. The strings you may include in the comma-separated list are: • schema — Update the schema used by Studio Assist when editing page definitions • `HTML` — Regenerate any JS or CSS files associated with this class • `SVG` — Regenerate any JS or SVG CSS files associated with this class The %ZEN.Component.object base class sets the default string to `"schema,HTML"` which is appropriate for Zen components and controls. The %ZEN.SVGComponent.svgComponent base class sets the default string to `"schema,SVG"` which is appropriate for Zen SVG components including meters. If you define MODULE and set POSTCOMPILATIONACTIONS to include either `HTML` or `SVG` then generated JavaScript is put in an external file, rather then being served inline with the page. You can use this feature to get the benefit of caching on the client and less overall overhead in serving content.

5.4 Custom Style

To define a new CSS style for a custom component, you can place an XData Style block in the subclass. Within this block place a `<style type="text/css">` tag and a closing `</style>` tag. Within the <style> element, place whatever CSS style definitions you like.

The built-in class %ZEN.Component.group includes the following XData Style block:

Class Member

```
XData Style
{
<style type="text/css">
/* @doc="Table used by groups." */
table.group {
  padding: 0px;
}

/* @doc="Cell within table used by groups." */
table.group td {
  padding: 0px;
}

/* @doc="Header within table used by groups." */
table.group th {
  padding: 0px;
}
</style>
}
```

Note the `/* @doc="text" */` syntax in the above example. When you provide a comment for an XData Style entry using this syntax, the Zen Style Wizard automatically includes a description of your style in its list of available styles. To display this list while editing a Zen class in Studio, choose **Tools > Templates > Templates** or press **CTRL-T** to display the list of Zen templates. Select the **Zen Style Wizard**; a list of all defined styles appears. This list is organized alphabetically by component name.

While viewing the list of styles in the Zen Style Wizard, you can select the radio button next to the name of the styles you want to edit and click OK. Templates for these styles appear in your XData Style block.

Important: If you are overriding existing styles in a subclass of one of the built-in Zen components, the Zen Style Wizard offers the most convenient way to select the styles you want to override.

The XData Style example above is a definition of styles for a built-in Zen component, %ZEN.Component.group. Suppose you wish to create a custom component with entirely new styles. In that case you want to create styles named appropriately for your component. A component called MyComponent might include an XData Style block like the one shown in the following example. Zen automatically includes this style definition in any page that uses <MyComponent> in its XData Contents block:

Class Member

```
XData Style
{
<style type="text/css">
/* @doc="Main style definition for MyComponent." */
.MyComponent {
    color: blue;
    background: yellow;
    white-space: nowrap;
}
</style>
}
```

Simply defining a XData Style block is not sufficient: the component class must actually *use* the newly defined CSS styles within the HTML that it generates. To accomplish this, you must provide a reference to the style in the **%DrawHTML** method that renders HTML for the custom component class. The following sample **%DrawHTML** method references the MyComponent style defined in the XData Style example above:

Class Member

```
Method %DrawHTML()
{
    &html<<div class="MyComponent">Message</div>>
}
```

By convention, any CSS style names you provide in an XData Style block should use a name that is related to the name of the component, as in the example above. This helps to avoid conflicts (which cannot be detected in advance due to the nature of CSS) and makes it easier for users to determine how to override styles. A component should never redefine a style defined by another component, or define an element-wide style selector (such as “input” or “table”), as this causes interference with other components.

Important: To review the order of precedence rules for CSS styles defined in component, page, and application classes, see the section “Cascade of Styles” in the “Zen Style” chapter of *Using Zen*. Custom components are subject to these rules.

5.4.1 XData SVGStyle

Style does not apply to Zen SVG components. Zen SVG components use XData SVGStyle.

To define styles for a custom Zen SVG component class, you can provide an XData SVGStyle block and add CSS statements between the `<style type="text/css">` tag and the closing `</style>` tag. For example:

Class Member

```
XData SVGStyle
{
<style type="text/css">
.customSVGComponent {
  fill: url(#myGrad);
  stroke: black;
  stroke-width: 2px;
}
</style>
}
```

5.4.2 XData SVGDef

The XData SVGStyle example above is from the class ZENTest.customSVGComponent in the SAMPLES namespace. It refers to a color `myGrad` as the fill color for the `customSVGComponent` shape.

`myGrad` is defined in the same class, but in a separate block called XData SVGDef. This block defines `myGrad` as a shaded gradient from blue to red:

Class Member

```
XData SVGDef
{
<defs>
<linearGradient id="myGrad" x1="0%" y1="0%" x2="0%" y2="100%">
<stop offset="0%" style="stop-color:darkblue" />
<stop offset="30%" style="stop-color:#FF00FF" />
<stop offset="70%" style="stop-color:#FF00FF" />
<stop offset="100%" style="stop-color:darkred" />
</linearGradient>
</defs>
}
```

5.4.3 Zen Color Definitions

The built-in class %ZEN.SVGComponent.svgPage provides an XData SVGDef block with several useful color definitions:

- The glow-silver color produces a metallic look for instrumentation items such as the <speedometer> meter.
- Meters such as the traffic light use glow-red, glow-yellow, and glow-green for indicator lamps.
- Lamp colors glow-blue, glow-purple, glow-orange, and glow-teal are also available.

<svgFrame> always references this class, or some subclass of it, through its *svgPage* attribute, so these colors are always available to any SVG component.

5.5 Custom Properties

Since a component is a class, it may define properties in addition to those it inherits. The definition of these properties influence how they work on the server, how they can be used within the component's corresponding JavaScript class definition, and how they can be specified within an XML representation of the page.

5.5.1 Naming Conventions

There are case-sensitive naming conventions for properties in Zen classes. For details, see the section "Zen Naming Conventions" in the "Zen Tutorial" chapter of *Using Zen*. These conventions may be summarized as follows:

- Client-only properties: `myProperty`
- Server-only properties (the % is essential): `%MyProperty`
- Properties that invoke an event handler via Javascript: `onmousedown`
- Properties that identify a server-side callback method: `OnCreateDataSet`
- Other properties: `myProperty`

5.5.2 XML Projection

All Zen components are derived from the %XML.Adaptor class and use its mechanisms for defining their XML behavior. If you would like more detail about this, see *Projecting Objects to XML*. In particular, for the XMLPROJECTION parameter introduced in this topic, see the section "Controlling the Projection for Simple Properties," which contains a table that describes the effects of each possible value of XMLPROJECTION: "attribute", "element", "wrapped", "none", and more.

All simple properties of Zen components should have an XMLPROJECTION value of "attribute". If you do not want to make a property visible via XML, set its XMLPROJECTION parameter to "none" as in the following example:

```
Property beeswax As %ZEN.Datatype.string(XMLPROJECTION="none");
```

Datatypes from the %ZEN.Datatype package use a default XMLPROJECTION value of "attribute" so that they are projected as XML attributes. Zen datatypes offer many conveniences when you are defining properties in a custom component class. For details, see the section "Datatype Classes."

5.5.3 setProperty Method

If you add a property to a custom component class, you must override the **setProperty** method in the subclass to implement cases for each property that you have added to the class. You can default to the superclass method for properties of the superclass. There is an example of this practice in this chapter. See the sample component class in the section "Helper Methods for %DrawHTML."

5.5.4 Datatype Parameters

All Zen classes inherit (from %ZEN.componentParameters) the following set of property parameters, which you can apply to any property you add to a Zen class. The property does not need to have a Zen datatype to use these parameters:

- ZENCLIENTONLY

- ZENENCRYPT
- ZENEXPRESSION
- ZENLOCALIZE
- ZENSETTING

5.5.4.1 ZENCLIENTONLY

There are cases when a Zen component might define properties whose value only makes sense within the client environment. For example, the *window* property of %ZEN.Component.object, or the *svgGroup* property used by SVG components. If the property parameter ZENCLIENTONLY is set to 1 (true) this indicates that a given property should be part of the client object model, but is never synchronized with any server-side changes to the object, and is not included within the serial state of the object. The ZENCLIENTONLY parameter value is an internal notation and is typically not required for use by Zen developers.

5.5.4.2 ZENENCRYPT

ZENENCRYPT is not a data encryption tool. It is used to prevent a property that controls the behavior of a component from being manipulated on the client. The best example is the *showQuery* property of %ZEN.Component.tablePane. ZENENCRYPT is set to true for this property so that no one can change the property in JavaScript on the client, and then be able to see the query being executed in the table. It is a protection mechanism in Zen to prevent modification of a component's critical behaviors.

5.5.4.3 ZENEXPRESSION

When a property has ZENEXPRESSION to 1 (true) this indicates that the property can interpret a Zen #()# expression to get its value at runtime. For details, see the "Zen Runtime Expressions" section in the chapter "Zen Application Programming."

Important: You cannot cause a property to support runtime expressions by setting `ZENEXPRESSION=1`. Built-in Zen component classes provide `ZENEXPRESSION=1` to *indicate* that the property supports runtime expressions, not to enable it to do so.

5.5.4.4 ZENLOCALIZE

Any Zen property that has its ZENLOCALIZE parameter set to 1 (true) automatically generates a message dictionary entry that can be used to translate the value of the property into another language. For details, see the chapter "Zen Localization."

It is a good practice to localize any string-valued properties that contain messages that you want to display to communicate with the Zen application user. It is convenient to give these properties the Zen datatype %ZEN.Datatype.caption, which always has ZENLOCALIZE set to 1, but you can also set ZENLOCALIZE directly by including `ZENLOCALIZE=1` in the property definition. Unlike the restriction on ZENEXPRESSION, setting the datatype parameter `ZENLOCALIZE=1` actually *enables* the property to be automatically localized.

The ZENLOCALIZE parameter:

- Applies only if the class that uses it also defines a localization domain by providing a value for the DOMAIN class parameter.
- Applies only to code that Zen generates from the XML description within XData Contents in the page class. If you bypass XData Contents and work with Zen pages programmatically, then you are responsible for handling the equivalent localization tasks, including calls to $$$Text macros. See the chapter "Zen Localization."

5.5.4.5 ZENSETTING

Set ZENSETTING to 1 (true) to specify that the property should be considered as a “setting” within the client class definition. This means that:

- The property becomes visible to Zen utilities such as the Control Test page
- You can observe and modify its value using the client-side **getProperty** and **setProperty** methods

Every datatype class in the %ZEN.Datatype package already sets ZENSETTING to 1. When you use the %ZEN.Datatype classes, you must set ZENSETTING to 0 for any properties that you do *not* want to be treated as settings. For example, properties that are of type %ZEN.Datatype.list or are projected as an object type on the client do not behave correctly with **getProperty** and **setProperty**, which treat the property as a string.

5.5.5 Datatype Classes

For convenience, Zen provides a set of datatype classes that you can use when defining properties in Zen classes. The datatype classes are in the %ZEN.Datatype package. The following table lists them.

When you define new Zen classes, you are free to use any datatype classes you wish. Using the %ZEN.Datatype classes makes it easier to understand the purpose of a property in the context of a web application. Also, in many cases datatype classes automatically enable features that you would otherwise have to encode yourself, including easy localization, event handler conventions, and language-independence for Boolean values and list arrays.

Fundamentally, all Zen datatypes classes are strings. All of the datatypes listed in the following table share the same base class, %ZEN.Datatype.datatype, whose value is defined as type %Library.String. As you can see from the entries in the table, each of these strings is interpreted by its subclass in a uniquely useful way.

Table 5–5: Datatype Classes

Class	Description
align	HTML horizontal alignment value. Possible values for this are “left”, “right”, and “center”. For vertical alignment values, see valign.
boolean	Boolean value: It can have the text value "true" or "false" in XData Contents. When accessed programmatically from ObjectScript, Caché Basic, or Caché MVBasic code running on the server, it can have the value 1 or 0. When accessed programmatically from JavaScript code running on the client, it can have the value true or false.
caption	String that is automatically added to the set of localized text resources when this property is initialized by an XData Contents block, as long as the page has defined a localization DOMAIN and its ZENLOCALIZE parameter is set to 1. For details, see the chapter “Zen Localization.”
classMember	Name of a server-side class member (such as a property name or method name). This class defines a datatype parameter, MEMBERTYPE, that indicates the type of class member. When using this datatype to define a property, you must also provide a value for its MEMBERTYPE parameter. Possible values for MEMBERTYPE are “PROPERTY”, “METHOD”, “QUERY”, “INDEX”, or “XDATA”.
className	Name of a server-side class.
color	CSS color value.
cssClass	Name of a CSS style class.
csv	Comma-separated list of values such as “John,Paul,George,Ringo”

Class	Description
delegator	Name of a server-side method within the current page class that is used as a callback method. For example, the Zen <html> component has an *OnDrawContent* attribute that specifies the name of a server-side callback method that provides HTML content by using &html or by using the WRITE command. If defined, this callback is invoked on the server whenever this component is drawn. In the underlying class %ZEN.Component.html, *OnDrawContent* is defined to be of type delegator.
eventHandler	JavaScript expression to be executed on the client in response to a client-side event. As an event handler, this JavaScript expression is expected to invoke a client-side method that is the “handler” for this event. For example, the Zen <button> component has an *onclick* attribute that specifies what should happen when a user clicks on the control. In the underlying class %ZEN.Component.button, *onclick* is defined to be of type eventHandler. You must use the eventHandler type if you wish to use the %GetEventHandlers helper method to access the event handler from %DrawHTML. For details, see the section “Custom Methods.”
expression	Server-side ObjectScript expression.
float	Floating point numeric value.
glvn	Name of a Caché global (multidimensional array) such as “^myGlobal”.
html	String of text that is marked up using HTML.
id	The id value for a component. It must not be used for any other purpose.
integer	Integer value.
length	HTML length value. For example: “5” or “5%”.
list	On the server a list represents a set of items as a piece-delimited string. On the client the list is converted to a JavaScript array. This datatype has a DELIMITER parameter, the value of which is used to delimit the server representation of the list. The default DELIMITER value is $C(5). Page properties that are of type %ZEN.Datatype.list or are projected as any "object" type on the client do not behave correctly with **getProperty** and **setProperty**, unless you set ZENSETTING to 0.
name	The name value for a component. It must not be used for any other purpose.
resource	Name of a Caché resource. If you are not familiar with Caché resources, see the “Assets and Resources” chapter in the *Caché Security Administration Guide* .
script	Client-side JavaScript expression.
sql	SQL statement. By default, properties of this type are encrypted when sent to the client (the ZENENCRYPT parameter is set to 1).
string	String with a default MAXLEN of 250. You can reset the MAXLEN value.
style	CSS style statement. For example: `"color:red; background: yellow;"`
svgStyle	SVG CSS style definition. Styles within SVG are CSS compliant, but there is a different set of styles available, so this datatype designates them.
uri	URI value. For example: “http://www.intersystems.com/zen”

Class	Description
valign	HTML vertical alignment value. Possible values are “top”, “bottom”, and “middle”. For horizontal alignment values, see align.
value	A value to be used as the value of an HTML control.

5.6 The %DrawHTML Method

Each custom component class must provide an implementation of the **%DrawHTML** method. **%DrawHTML** is responsible for providing the HTML (or SVG) needed to create the client-side visual representation of a Zen component. The method is called when the page containing the component is first served. It may subsequently be called if the page needs to dynamically refresh the contents of the component, such as when the query for a <tablePane> is re-executed.

The basic operation of the **%DrawHTML** method is very simple: Any output that this method produces is served up as HTML within the browser. You can output from **%DrawHTML** using the WRITE command, as follows:

Class Member

```
Method %DrawHTML()
{
        Write "This is some <b>HTML!</b>",!
}
```

As an alternative to WRITE, you can use the ObjectScript syntax for embedded HTML statements. This syntax uses &html followed by HTML statements enclosed in angle brackets <> as in the following example:

Class Member

```
Method %DrawHTML()
{
        &html<This is some <b>HTML!</b>>
}
```

Any output from **%DrawHTML** is enveloped by the component’s enclosing element, as supplied by the Zen framework.

What makes this convention powerful is that **%DrawHTML** is an instance method of the component class: it can execute logic and has access to the component’s properties and methods, as well as the underlying Caché database. It supports ObjectScript expression, CSP #()# syntax, and Zen #()# expression syntax. The following simple example uses a number of these features:

Class Member

```
Method %DrawHTML()
{
        #; draw items as specified by myCount
        For i=1:1:..myCount {
                &html<This is item #(i)#.<br/>>
        }
}
```

5.6.1 Helper Methods for %DrawHTML

The Zen framework defines three server-side helper methods for use in the **%DrawHTML** method for custom components: %MakeId, %Attr, and %GetEventHandlers. The class code for <button> shows one way to use these methods effectively in **%DrawHTML**:

Class Definition

```
Class %ZEN.Component.button Extends control
{

Parameter DEFAULTCONTROLCLASS = "button";

/// Caption displayed for this button.<br>
/// This is a localized value.
Property caption As %ZEN.Datatype.caption;

/// defines style sheet used by this component
XData Style
{
<style type="text/css">
/* @doc="Style for button (input)." */
.button {
}
</style>
}

Method %DrawHTML()
{
  Set disabled = $S(..disabled:"disabled",1:"")
  Set tIgnore("onchange") = ""
  &html<
  <input type="button" class="#(..controlClass)#"
  id="#(..%MakeId("control"))#" #(..%Attr("title",..title))#
  #(..%Attr("name",..name))# #(..%Attr("value",..caption))# #(disabled)#
  #(..%Attr("style",..controlStyle))# #(..%GetEventHandlers(.tIgnore))#>
  >
}

/// This method fills in reasonable default values for
/// this control. Used by tools (such as Control Tester) to
/// dynamically create controls.
Method %SetDefaultValues()
{
  Set ..caption = "Button"
}

/// Set the value of a named property.
ClientMethod setProperty(property, value, value2) [ Language = javascript ]
{
  switch(property) {
  case 'caption':
    this.caption = value;
    var el = this.findElement('control');
    if (el) {
      el.value = this.caption;
    }
    break;
  case 'value':
    // do not set control value; just internal value
    this.value = value;
    break;
  default:
    // dispatch
    return this.invokeSuper('setProperty',arguments);
  }
  return true;
}
}
```

Important: This example is close, but not identical, to the built-in class %ZEN.Component.button.

The following topics discuss how each of the **%DrawHTML** helper methods is used:

- %MakeId — Generate the id value for the HTML element that displays the component on the page. Use a consistent naming scheme so that the client-side method **findElement** can find the HTML element on the page.
- %Attr — Assign a value to an attribute of the HTML element that displays the component on the page. This method may be called repeatedly to assign values to all the HTML attributes required to render the component.
- %GetEventHandlers — Retrieve all of the event handling information from the component definition, so that the HTML representation of the component can correctly match each user event with the code that should execute.

5.6.2 Identifying HTML Elements

The **%MakeId** server-side method ensures uniqueness for the id of every HTML element on the output page. The example in the "Helper Methods for %DrawHTML" section calls **%MakeId** from embedded HTML to set the id for the HTML <input> element as follows:

```
id="#(..%MakeId("control"))#"
```

%MakeId concatenates the string provided by the caller with the underscore character "_" followed by the unique sequential number that Zen assigns to every component it places on the page. It then gives this identifier to Zen to use as the value for the id attribute in the enclosing for the element. Additionally, if the component is part of a repeating group, Zen adds a further underscore character "_" followed by the tuple number.

Thus, if you view the source of any generated Zen page you see enclosing elements with generated id values like `spacer_23` in the following example:

XML

```
<div class="spacer" id="spacer_23" style="width:20px;"/>
```

The following more complex excerpt shows how Zen provides unique HTML id values for each element in a repeating group of radio buttons. This excerpt is part of the page that results when Zen renders the class ZENDemo.FormDemo from the SAMPLES namespace:

```
<tr valign="top">
<td style="padding: 4px; padding-left: 5px; padding-right: 5px;" >
<span id="zenlbl_26" class="zenLabel"  >Marital Status:</span>
<div class="zendiv" id="MaritalStatus" >
<input type="hidden" id="hidden_26" name="$V_MaritalStatus" value="">
<span class="radioSetSpan"><input type="radio" id="textRadio_1_26"
      name="r26" value="S" onclick="zenPage.getComponent(26).clickItem(1);">
   <a class="radioSetCaption" id="caption_1_26" href=""
      onclick="javascript:zenPage.getComponent(26).clickItem(1);return false;">
      Single</a> </span>
<span class="radioSetSpan"><input type="radio" id="textRadio_2_26"
      name="r26" value="M" onclick="zenPage.getComponent(26).clickItem(2);">
   <a class="radioSetCaption" id="caption_2_26"  href=""
      onclick="javascript:zenPage.getComponent(26).clickItem(2);return false;">
      Married</a> </span>
<span class="radioSetSpan"><input type="radio" id="textRadio_3_26"
      name="r26" value="D" onclick="zenPage.getComponent(26).clickItem(3);">
   <a class="radioSetCaption" id="caption_3_26"  href=""
      onclick="javascript:zenPage.getComponent(26).clickItem(3);return false;">
      Divorced</a> </span>
<span class="radioSetSpan"><input type="radio" id="textRadio_4_26"
      name="r26" value="W" onclick="zenPage.getComponent(26).clickItem(4);">
   <a class="radioSetCaption" id="caption_4_26"  href=""
      onclick="javascript:zenPage.getComponent(26).clickItem(4);return false;">
      Widowed</a> </span>
<span class="radioSetSpan"><input type="radio" id="textRadio_5_26"
      name="r26" value="O" onclick="zenPage.getComponent(26).clickItem(5);">
   <a class="radioSetCaption" id="caption_5_26"  href=""
      onclick="javascript:zenPage.getComponent(26).clickItem(5);return false;">
      Other</a> </span>
</div>
</td>
</tr>
```

The client-side equivalent for **%MakeId** is a JavaScript method called **makeId**. The custom meter component examples use **makeId**.

5.6.3 Finding HTML Elements

When you are writing client-side code and need to access a specific HTML element on the page, you can use the **findElement** method. Its argument is the value that you assigned to the *id* attribute when placing the component on the page. **findElement** combines this string with the sequential identifier for the element on the output page to generate the appropriate HTML id

for the element on the output page. It uses this information to obtain a pointer to the component object within the page object model for the rendered page, then returns that pointer so that you can use properties and methods of the component object.

Ordinary Zen components inherit **findElement** from %ZEN.Component.object. For SVG components you must use **findSVGElement** from %ZEN.SVGComponent.svgComponent. The custom meter component examples use **findSVGElement**.

findElement and **findSVGElement** work correctly only if you have assigned HTML id values using **%MakeId** (on the server) or **makeId** (on the client).

5.6.4 Setting HTML Attribute Values

The **%Attr** server-side method assigns a value to an attribute of the HTML element that displays the component on the page. This method may be called repeatedly to assign values to all the HTML attributes required to render the component. The following sample **%DrawHTML** method uses **%Attr** in rendering the <button> component:

Class Member

```
Method %DrawHTML()
{
  Set disabled = $S(..disabled:"disabled",1:"")
  Set tIgnore("onchange") = ""
  &html<<input type="button" class="#(..controlClass)#"
               id="#(..%MakeId("control"))#" #(..%Attr("title",..title))#
                   #(..%Attr("name",..name))# #(..%Attr("value",..caption))#
                   #(disabled)# #(..%Attr("style",..controlStyle))#
                   #(..%GetEventHandlers(.tIgnore))#>>
}
```

Suppose the value of the <button> *controlStyle* attribute is currently `myActionStyle`. As the **%DrawHTML** method outputs HTML, an expression in this format:

```
#(..%Attr("style",..controlStyle))#
```

Generates the following output:

```
style="myActionStyle"
```

5.6.5 Attaching Event Handlers to HTML Elements

The **%GetEventHandlers** server-side method gets all the event handler attributes for a given component and uses them to write out any event handler attributes for the HTML that displays that component in the output page.

Important: **%GetEventHandlers** can only find event handler attributes whose underlying property has been defined using the datatype class %ZEN.Datatype.eventHandler.

For a list of the event handler attributes that you can specify for a control component on a Zen form, see the section "Control Attributes" in the "Zen Controls" chapter of *Using Zen Components*. The types of events covered by this list include mouse movements, mouse clicks, or key clicks. Other types of Zen component have event handlers, but Zen controls offer the single largest pool for comparison.

Preceding chapters have explained that to set up an event handler for a built-in Zen component you must specify a JavaScript expression as the value of the corresponding event handler attribute. Generally this JavaScript expression invokes a client-side method that serves as the "handler" for this event. Of course, if you set up the component this way you must also write the client-side method that is being invoked.

For custom components, you must take additional steps to connect each user event with its appropriate handler. This connection takes place in the **%DrawHTML** method for the custom component. Consider the **%DrawHTML** method for the <image> component:

Class Member

```
Method %DrawHTML()
{
  #; handle onclick directly
  Set tIgnore("onclick")=""
  Set tIgnore("onchange")=""

  #; select image to display
  Set tSrc = ..src
  If (..streamId '= "") {
    Set tSrc = ##class(%CSP.Page).Link(
    "%25CSP.StreamServer.cls?STREAMOID="_$ZCVT(..streamId,"O","URL"))
  }

  #; disabled logic
  Set tSrc = $Case(..disabled,0:tSrc,:$S(..srcDisabled="":tSrc,1:..srcDisabled))

  &html<<img id="#(..%MakeId("control"))#"
             #($S(..onclick="":"",1:"class=""imageLink"""))#
             #(..%Attr("src",tSrc))#
             #(..%Attr("title",..title))#
             #(..%Attr("width",..width))#
             #(..%Attr("height",..height))#
             #(..%GetEventHandlers(.tIgnore))#
             onclick="zenPage.getComponent(#(..index)#).imageClick();"/>>
}
```

In the above example, some interesting things take place with regard to event handlers. The following line tells **%GetEventHandlers** to ignore any value provided for the *onchange* event handler, even if it discovers that there is such a value:

```
Set tIgnore("onchange") = ""
```

You must provide one such line for each event handler that you want to handle directly in **%DrawHTML**, as shown for *onclick* and *onchange* in the example above. Then, at the end of **%DrawHTML**, pass the variable *tIgnore* to **%GetEventHandlers** by reference, as shown in the example:

```
#(..%GetEventHandlers(.tIgnore))#
```

The **%DrawHTML** example above asks **%GetEventHandlers** to ignore *onclick* and *onchange*, but for different reasons in each case: <image> ignores *onchange* because it does not accept user input. <image> ignores *onclick* because it is designed to handle all *onclick* events in the same way, without permitting the Zen programmer to specify a different behavior in the page class. Therefore, it provides code in **%DrawHTML** to handle this event directly. Any event handlers that are not expressly ignored by **%DrawHTML** in this way are written out to the page exactly as defined by the Zen programmer.

Now consider this expression from the example above:

```
#($S(..onclick="":"",1:"class=""imageLink"""))#
```

This expression uses the ObjectScript $SELECT function to return the first true expression it encounters in a list of expressions. Looking at the expressions provided: If the <image> component's *onclick* attribute value is an empty string, this statement writes an empty string. Otherwise, this statement formats the HTML element on the output page by writing out the following string:

```
class="imageLink"
```

The following expression directs the client-side HTML page to invoke the component's **imageClick** method in response to any user mouse clicks on the corresponding HTML element.

```
onclick="zenPage.getComponent(#(..index)#).imageClick();"
```

The **imageClick** method is a client-side JavaScript method in the <image> component class. It uses the JavaScript helper method **zenInvokeCallbackMethod** to invoke *onclick* callback method for this <image> component. The **imageClick** method looks like this:

Class Member

```
ClientMethod imageClick() [ Language = javascript ]
{
  if (!this.disabled) {
    // invoke callback, if present
    zenInvokeCallbackMethod(this.onclick,this,'onclick');
  }
}
```

Where the **zenInvokeCallbackMethod** arguments are as follows:

- `this.onclick` — The component's *onclick* attribute value (a JavaScript expression that invokes a client-side JavaScript method)
- `this` — A pointer to the component object
- `'onclick'` — The HTML name for the event (must be quoted)

This completes the connection between the HTML element, the event, and its handler. You must provide similar conventions for any event that you intend your custom component to support.

5.6.6 HTML for Dynamic Components

%DrawHTML works differently for components that change in response to runtime information or that are generated using SVG.

The <calendar> is an example of a component that writes no static HTML at all. Its **%DrawHTML** method simply sets the values of properties such as the currently selected month and date, in preparation for the more complex JavaScript method **renderContents** that is invoked whenever the page needs to redraw the calendar. **renderContents** and its helper methods generate an array of dynamic HTML statements based on the user's current selections of calendar month and date. For details, examine the class code for %ZEN.Component.calendar.

5.7 Custom Methods

Since a component is a class, it may define methods in addition to those it inherits. The definitions of these methods influence where and how they can be used. These rules are exactly the same for methods in Zen component classes as they are for methods within a Zen page. The following table summarizes the conventions for method in Zen classes. For details, see the "Zen Page Methods" section in the chapter "Zen Application Programming."

Table 5–6: Component Class Method Conventions

Callable From	Runs On	Code Language	Keywords	Naming Convention
Client	Client	JavaScript	ClientMethod [Language = javascript]	`myMethod`
Client	Server (but can send back code that runs on the client)	ObjectScript, Caché Basic, Caché MVBasic	[ZenMethod]	`MyMethod`
Server	Server	ObjectScript, Caché Basic, Caché MVBasic	—	`%MyMethod`

The most important method in your custom component class is the required server-side method %DrawHTML. The previous section describes how to work with this method. The following sections describe other methods that you might want to override in a custom component class. These include:

- %OnDrawEnclosingDiv
- %OnDrawTitleOptions
- Data Drag and Drop Methods

5.7.1 %OnDrawEnclosingDiv

If defined in a component subclass, this callback method, makes it possible for the subclass to inject additional HTML attributes into the component's enclosing element at runtime. **%OnDrawEnclosingDiv** has no input arguments. If implemented, the method should return a %String value. The string must include a leading space to avoid conflict with other attributes that may already be modifying the element.

5.7.2 %OnDrawTitleOptions

The .expando class offers a server-side callback method **%OnDrawTitleOptions** which, if defined in a subclass of .expando, provides a way to add content to the right side of the title bar when the <expando> has its *framed* attribute set to true. Any HTML written by **%OnDrawTitleOptions** is injected into the title bar of the <expando> when it is displayed. For details, see "<expando>" in the "Navigation Components" chapter of *Using Zen Components*.

5.7.3 Data Drag and Drop Methods

It is possible to provide values for the component attributes *onafterdrag*, *onbeforedrag*, *ondrag*, and *ondrop* each time you place a component on the page, as described in the "Drag and Drop" section of the "Zen Component Concepts" chapter of *Using Zen*. However, rather than requiring Zen application developers to set data drag and drop attributes per component, you can build the desired behavior into a custom control component. This way, when placing a component on the page, developers can simply choose the custom control, which already has the desired behavior.

If you want to customize data drag and drop for a custom control class, there is a group of methods in the base classes %ZEN.Component.component and %ZEN.Component.control that you can override. Each of these methods accepts one parameter, `dragData`, which is a pointer to a JavaScript object whose methods and properties are defined in the Zen client side library. These methods also call functions defined within the library, such as `ZLM.setDragAvatar`. For full details about any of these items, refer to the "Client Side Library" chapter in this book.

Note: Because the library is written in JavaScript, which does not facilitate generated class documentation, there is no Class Reference documentation for it in the InterSystems online documentation system. See the "Client Side Library" chapter for all details.

The following table traces the sequence of internal events when drag and drop occurs between a source Zen component and a target Zen component. During these events, certain JavaScript methods in the source and target component classes call certain JavaScript functions within the Zen client side library. The component methods are triggered automatically by the Zen framework. When you customize data drag and drop for a custom control class, you can change the details of what happens inside these methods when they are automatically invoked.

Table 5–7: Data Drag and Drop Sequence

	User	Client Side Library	Component Method
1	Starts the drag from the source component.		

	User	Client Side Library	Component Method
2		Calls the source component's **dragHandler** method.	
3			Creates a zenDragData object named dragData to hold the data transferred by the drag.
4			Invokes the source component's onbeforedrag event, if defined. If the onbeforedrag event returns false or if dragData.value has not been set, the drag is cancelled and dragHandler exits.
5			Calls the source component's **dragStartHandler** method. This is where an individual component's default drag handling is implemented. **dragStartHandler** can return false to cancel the drag.
6			Invokes the source component's ondrag event, if defined. This gives a component a chance to modify the drag data or cancel the drag. If the ondrag event return false, then the drag is cancelled.
7	Releases the drop over the target component.		
8		Calls the target component's dropHandler method	
9			Invokes the target component's ondrop event, if defined. If the ondrop event returns false, then the dropHandler method exits.
10			Calls the target component's **dropStartHandler** method. This is where an individual component's default drop handling is implemented. **dropStartHandler** can return false to cancel the drop.
11			Calls the source component's dragNotifyHandler method to notify the source component that the drag is complete.
12			The dragNotifyHandler calls the source component's **dragFinishHandler** method and invokes the source component's onafterdrag event, if defined.

There are several methods in the base classes %ZEN.Component.component and %ZEN.Component.control that are designed for you to override. The next several sections describe them. Other methods in the previous table are part of the Zen framework, and InterSystems recommends that you do not attempt to customize them. The methods designed for override are:

- getDragData

- dragStartHandler
- dragFinishHandler
- dropStartHandler

5.7.3.1 getDragData

getDragData is a JavaScript method that is called automatically each time the user begins a drag operation on any control or <image> component. Other components do not support the **getDragData** method. The base class for control components, %ZEN.Component.control, implements a base version of **getDragData** that looks like the following example. Its purpose is to set the `text` and `value` fields in the `dataDrag` object.

Class Member

```
ClientMethod getDragData(dragData) [ Language = javascript ]
{
  dragData.value = this.getValue();
  if (null != this.text) {
    // if there is a text property, use it as the text value
    dragData.text = this.text;
  }
  else {
    dragData.text = dragData.value;
  }
  return true;
}
```

Subclasses of %ZEN.Component.control can override **getDragData** to provide customized behavior. It does not matter what happens inside the method as long as it sets values for `dataDrag.text` and `dataDrag.value` before it returns true. The next example shows how the class %ZEN.Component.image overrides **getDragData**. This method assigns the string value of the <image> *text* attribute as both the logical value and the display value to be dragged:

Class Member

```
ClientMethod getDragData(dragData) [ Language = javascript ]
{
  dragData.value = this.text;
  dragData.text = this.text;
  return true;
}
```

5.7.3.2 dragStartHandler

dragStartHandler is a JavaScript method that is automatically called whenever a drag operation is started within a component. The base class for control components, %ZEN.Component.control, implements a base version of the **dragStartHandler** method, which looks like the following example. This **dragStartHandler** method does the following:

- Calls **getDragData** to set `dragData.text` and `dragData.value`
- Creates an avatar (`icon`) to represent the field as it is dragged across the page
- Calls a client side library function (`ZLM.setDragAvatar`) to use this icon

Class Member

```
ClientMethod dragStartHandler(dragData) [ Language = javascript ]
{
  // get drag data
  if (!this.getDragData(dragData)) {
    return false;
  }

  // avatar
  var icon = this.getEnclosingDiv().cloneNode(true);
  icon.style.position="absolute";
  icon.style.border ="1px solid darkgray";
  icon.style.background ="#D0D0F0";
  ZLM.setDragAvatar(icon);

  return true;
}
```

Subclasses of %ZEN.Component.control can override **dragStartHandler** to provide customized behavior. For example, the class %ZEN.Component.abstractListBox overrides **dragStartHandler** as shown in the next example. This example does the following:

- Omits any call to **getDragData**.
- Acquires a pointer (`dragItem`) to the specific list item where the drag began
- Finds the exact pointer (`anchor`) to this item within the list control
- Sets the `sourceItem`, `value`, and `text` fields in the `dragData` object
- Creates an avatar (`icon`) to represent the field as it is dragged across the page
- Calls a client side library function (`ZLM.setDragAvatar`) to use this icon

Class Member

```
ClientMethod dragStartHandler(dragData) [ Language = javascript ]
{
  var ok = false;
  var dragItem = this._dragSource;
  if (null != dragItem) {
    delete this._dragSource;
    var anchor = this.findElement('item_' + dragItem);
    if (anchor) {
      dragData.sourceItem = dragItem;
      ok = true;
      dragData.value = this.getOptionValue(dragItem);
      dragData.text = this.getOptionText(dragItem);

      // avatar
      var icon = anchor.cloneNode(true);
      icon.style.position ="absolute";
      icon.style.width = this.getEnclosingDiv().offsetWidth + 'px';
      icon.style.border = "1px solid darkgray";
      ZLM.setDragAvatar(icon);
    }
  }
  return ok;
}
```

5.7.3.3 dragFinishHandler

dragFinishHandler is a JavaScript method that is automatically called whenever a drag operation that was started within this component is completed. This occurs as soon as the user releases the mouse button after having held it down for the "drag" operation. **dragFinishHandler** provides an opportunity to tie up any loose ends that may remain for the source component at this point.

5.7.3.4 dropStartHandler

dropStartHandler is a JavaScript method that is automatically called whenever a drop operation is started within a component. The base class for control components, %ZEN.Component.control, implements a base version of **dropStartHandler** that looks like the following example. It sets the value of the target control to `dragData.toString()`. This value was previously set by the activities of other drag and drop methods in the class. Generally it is the value acquired from the source control.

Class Member

```
ClientMethod dropStartHandler(dragData) [ Language = javascript ]
{
  this.setValue(dragData.toString());
  return true;
}
```

Subclasses of %ZEN.Component.control can override **dropStartHandler** to provide customized behavior. For example, the class %ZEN.Component.listBox overrides the base control **dropStartHandler** as shown in the next example. This example does the following:

- Gets the `value` and `text` fields from the `dragData` object.
- If this <listBox> is receiving a drop from some other component:

  Call `appendOption` to:

  - Create a new list option that has the `value` and `text` fields as its logical and display values
  - Append this new option to the <listBox>
  - Redisplay the <listBox>

- If this <listBox> is receiving a drop from itself:
  - Call the client side library function `ZLM.getDragInnerDestination` to retrieve the identifier for the innermost enclosing where the drag operation ended.
  - Use this to calculate the appropriate `dragData.targetItem` identifier.
  - Call `moveOption` to:
    - Move the list option at position `dragData.sourceItem` to the position identified `dragData.targetItem`
    - Shift the other list options to make room
    - Redisplay the <listBox>

Class Member

```
ClientMethod dropStartHandler(dragData) [ Language = javascript ]
{
  var value = dragData.value;
  var text = dragData.text;

  if (this != dragData.sourceComponent) {
    // drag from another component: append
    this.appendOption(value,text);
  }
  else {
    // move item within this list
    var tgtId = ZLM.getDragInnerDestination().id;
    var tgtIndex = -1;
    if (tgtId && tgtId.indexOf('item')!=-1) {
      tgtIndex = tgtId.split('_')[1];
    }
    dragData.targetItem = tgtIndex;
```

```
    var srcIndex = dragData.sourceItem;
    this.moveOption(srcIndex,tgtIndex);
  }
  return true;
}
```

5.8 Sample Code

The following is an example of a custom component that defines a title bar. This example is from the ZENDemo package in the SAMPLES namespace:

Class Definition

```
Class ZENDemo.Component.demoTitle Extends %ZEN.Component.component
{

 /// XML namespace for this component.
Parameter NAMESPACE = "http://www.intersystems.com/zendemo";

/// Domain used for localization.
Parameter DOMAIN = "ZENDemo";

/// Title displayed within this pane.
Property title As %ZEN.Datatype.caption;

/// Category displayed within this pane (above the title).
Property category As %ZEN.Datatype.caption
                  [ InitialExpression = {$$$Text("ZEN Demonstration")} ];

/// defines style sheet used by this component
XData Style
{
<style type="text/css">
.demoTitle {
  color: black;
  background: #c5d6d6;
  width: 100%;
  padding: 0px;
  border-bottom: 1px solid darkblue;
  font-size: 1.4em;
  font-family: verdana;
  text-align: center;
}
</style>
}

/// Draw the HTML contents of this component.
Method %DrawHTML()
{
  Set tCategory = ..category

  &html<<table class="demoTitle" border="0" cellpadding="0" cellspacing="0"
              width="100%">
    <tr>
      <td align="left" width="40">
        <img width="185" height="60" src="images/zentitle.jpg"/>
      </td>
    <td align="left" width="90%" style="padding-left:20px;">
    <div style="font-size: 0.6em;">#($ZCVT(tCategory,"O","HTML"))#</div>
    <div>#($ZCVT(..title,"O","HTML"))#</div></td>
    <td> </td></tr></table>>
}
}
```

The purpose of defining a custom component for a title bar is to provide a consistent look for every page in a web application. Page classes in the ZENDemo package provide countless examples of the correct way to reference a custom component. An XData Contents block that provides a correct reference looks like this:

Class Member

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
<page xmlns="http://www.intersystems.com/zen"
      xmlns:demo="http://www.intersystems.com/zendemo"
      title="Control Test">

  <demo:demoTitle id="title"
                  title="Zen Control Test Page"
                  category="Zen Test Suite" />

<!-- more page contents here -->

</page>
}
```

Where:

- The <page> element references the namespace defined for the custom component, and defines an alias for it, by providing the following attribute within the opening <page> element:

  `xmlns:demo="http://www.intersystems.com/zendemo"`

  This statement defines an alias `demo` for the full namespace name.

- The <demo:demoTitle> element references the demoTitle custom component. This reference has the form:

  <alias:component>

  where *alias* is the namespace alias defined in the <page> element (`demo` in this case) and *component* is the name of the custom component class (`demoTitle` in this case).

There are further examples of custom components, and references to these components, in the SAMPLES namespace. You can examine these classes in Studio. The examples of custom component classes include:

- ZENDemo.Component.demoMenu, a subclass of %ZEN.Component.composite
- ZENDemo.Component.sidebar, a subclass of %ZEN.Component.component
- ZENDemo.Component.bullet, a subclass of %ZEN.Component.object
- ZENTest.customComponent, a subclass of %ZEN.Component.control
- ZENTest.customSVGComponent, a subclass of %ZEN.SVGComponent.svgComponent

5.9 Creating Custom Meters

The following table is your checklist for building a custom meter component. To supplement the information in this table, see the odometer, clock, and other custom meter examples that have been posted on the Zen Community pages:

`http://www.intersystems.com/community/zen`

Table 5–8: Extending Zen with Custom Meters

Task	Description
Subclass	Extend %ZEN.SVGComponent.meter to create a new class.
renderMeter	Any subclass of %ZEN.SVGComponent.meter must override this method to render the SVG contents of the meter by making SVG calls from the meter class.

Task	Description
Parameters	(Optional) Provide class parameters, such as DEFAULTHEIGHT and DEFAULTVIEWBOXHEIGHT, and give them appropriate values.
Properties	(Optional) Provide properties, if needed. For example, an odometer might set a maximum number of digits for its display. A clock might override the *value* property from its base meter class with an ObjectScript expression that calculates the current time each time the meter is displayed.
XData SVGStyle	(Optional) Provide any CSS style definitions for the meter.
XData SVGDef	(Optional) Provide any SVG definitions for the meter.
setProperty	(Optional) You may override this method to provide unique ways to set certain properties of the meter. A clock example might do this for the *value* property, as described above, then defer to the superclass for setting all other properties.
Methods	(Optional) Provide supporting methods as needed for component behavior.
renderLabel	(Optional) Any subclass of %ZEN.SVGComponent.meter may override this method to set the x and y positions of the meter label.
XML Element	To add the component to a page, place it within that page's XData Contents block as an XML element.

6

Client Side Layout Managers

At runtime, Zen builds a page description on the server and ships it to the client, where the page description is unpacked, expanded, and displayed as specified. This is the classic browser-based web application that serves previously laid-out pages to remote users upon receiving their requests for display. In this case, the Zen application maintains full control over what it shows to the user.

Note: For more about this typical case, see the section "Zen Pages at Runtime" in the "Zen Client and Server" chapter of *Using Zen*.

Client side layout provides a useful exception to this typical case. This chapter explains how to add components to Zen pages that permit a user to initiate client side adjustments to the page layout *after* the page has been rendered on the client. That is, the user directly manipulates the page layout by clicking and dragging the mouse.

Under this model, the user interface runs in a web browser, but it is not confined to the strict request-response interchange of a web application. Rather, the browser is the means for delivering a dynamic user interface that might otherwise be written in a language such as Visual Basic. Zen supports this type of application by providing several options for client side layout management.

6.1 Using Active Groups to Manage Layout

The simplest option for client side layout management is to write Zen pages using the built-in *client side layout managers*. These built-in components are called *active group* components. Each of them is a Zen group component, with special capabilities to permit direct manipulation on the client side. Each active group is also a client side layout manager; the terms are interchangeable.

The built-in active groups are:

- <activeHGroup>
- <activeVGroup>
- <corkboard>, which may contain <dragGroup> components
- <desktop>, which may contain <dragGroup> components
- <snapGrid>, which may contain <dragGroup> components

This chapter describes how to use the built-in active groups. The next several sections describe:

- Vertical and Horizontal Active Groups
- Active Groups that Resize Proportionally

- Calculating Percentages for Three-Way Splits

If you want to customize active group behavior in a way that applies to all components within the group, you can use the Zen client side library module to write your own client side layout manager (active group). For a full reference guide, see the chapter "Client Side Library" in this book.

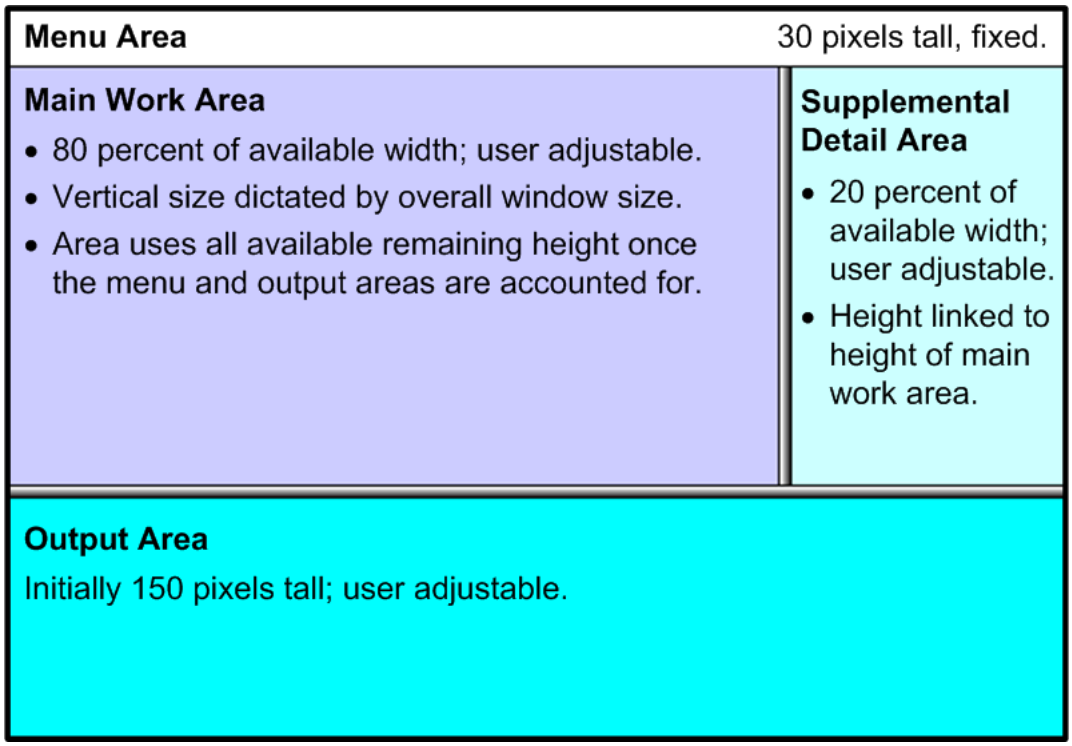
6.1.1 Vertical and Horizontal Active Groups

Suppose a designer wants to divide the screen into four regions:

- A fixed menu area across the top
- An output pane across the bottom
- A detailed information display area on the right
- A main work area that would expand to use all remaining window real estate once the other areas had been addressed

The designer also wants the user to be able to interactively adjust how much of the screen is devoted to the output and detail panes at runtime. Furthermore, the designer wants the layout to automatically adjust to changes in window size. That is, increasing the browser window to full screen mode would resize the main work and supplemental detail areas while leaving the size of the menu and output areas fixed. The following figure shows a conceptual view of this design:

Figure 6–1: Zen Page Layout Using Active Groups



The <activeVGroup> and <activeHGroup> are ideally suited for addressing these design objectives. Unlike <vgroup> and <hgroup> which are ultimately based on HTML tables, active groups use CSS and JavaScript to divide their available screen area into discrete subsections. This subdivision is always a *binary* split, resulting in top and bottom children for vertical divisions, or left and right for horizontal divisions.

This binary geometry is an implementation constraint, but not a design restriction. Active groups can be nested within one another to create the illusion of multi-way splitting of the available screen real estate. An example of this is shown in the

previous figure where there are three divisions along the vertical: the menu area at the top, the combined main and supplemental areas across the center, and the output area at the bottom.

You can provide the layout shown in the previous figure by using three nested active groups as shown in the following Zen page class:

Class Definition

```
Class MyApp.ActiveTest Extends %ZEN.Component.page
{
Parameter PAGENAME = "ActiveGroupLayout";

XData Style
{
<style type="text/css">
  #outerSplit {
    width:100%;
    height:100%;
    }
</style>
}

XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<page xmlns="http://www.intersystems.com/zen" layout="none">
  <activeVGroup id="outerSplit" noResize="true" handleThickness="1" split="30" >
    <group enclosingStyle="width:100%;height:100%;background:white;">
      <!-- Menu Area -->
    </group>
    <activeVGroup id="innerVSplit"  split="-150" >
      <activeHGroup id="horizontalSplit" split="80%" >
        <group enclosingStyle="width:100%;height:100%;background:#ccccff;">
          <!-- Main Work Area -->
        </group>
        <group enclosingStyle="width:100%;height:100%;background:#ccffff;">
          <!-- Supplemental Detail Area -->
        </group>
      </activeHGroup>
      <group enclosingStyle="width:100%;height:100%;background:cyan;">
        <!-- Output Area" -->
      </group>
    </activeVGroup>
  </activeVGroup>
</page>
}

ClientMethod onloadHandler() [ Language = javascript ]
  {
    ZLM.refreshLayout();
  }

ClientMethod onlayoutHandler() [ Language = javascript ]
  {
    ZLM.notifyResize(document.body);
  }
}
```

In the previous code example:

- The <activeVGroup> called `outerSplit` divides the window vertically into two regions. The top area is 30 pixels tall, is not user adjustable, and is partitioned off from the bottom area by a divider that is 1 pixel wide. The bottom area uses all but 31 pixels of the window's height; 31 is the top height plus the handle thickness. The top area contains the label identifying the menu area. The bottom area contains an additional <activeVGroup>.
- The <activeVGroup> called `innerVSplit` again divides the window vertically where one region is 150 pixels tall. In this case, however, the split property is negative, indicating that the property is referring to space reserved for the *bottom* area. By default, the split is user adjustable and displays an adjustment bar of sufficient thickness to be easily grabbed by the mouse pointer.
- The constraints set up by the two <activeVGroup> elements have the effect of reserving 30 pixels at the top of the window, 150 pixels at the bottom, and a few (actually, 8) for adjustment and partition bars. All remaining vertical pixels are funneled into the first child of the `innerVSplit` group. This child is an <activeHGroup> that partitions

the middle band of the window into the main work area and the supplemental detail area. This split is an 80% division of the available window width and is user adjustable by default.

6.1.2 Active Groups that Resize Proportionally

<activeVGroup> and <activeHGroup> allow "size to fit" operations on their contents. The following four steps are required to enable this functionality. Once this is set up, the groups automatically expand to fill the window on initial page load and adjust their sizes automatically in response to window resize events:

1. The <page> component must have its *layout* attribute set to `"none"`:

   ```
   <page xmlns="http://www.intersystems.com/zen" layout="none">
   ```

   This shifts responsibility for page layout from the server to the browser.

2. The outermost active group in the XData Contents block must have its CSS width and height explicitly set to `100%`. This can be set using the *enclosingStyle* attribute for the <activeVGroup> or <activeHGroup>. Alternatively you can do it as shown in the previous example of a Zen page class:

   - Define a CSS style rule:

     Class Member

     ```
     XData Style
     {
     <style type="text/css">
       #outerSplit {
         width:100%;
         height:100%;
         }
     </style>
     }
     ```

   - ...and apply this rule to the outermost active group using the *id* attribute:

     Class Member

     ```
     XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
     {
     <page xmlns="http://www.intersystems.com/zen" layout="none">
       <activeVGroup id="outerSplit" noResize="true" handleThickness="1" split="30" >
         <!-- Everything else -->
       </activeVGroup>
     </page>
     }
     ```

3. The following must be true *all the way down* the hierarchy of group components, from the top-level <page> container down to any group that you wish to resize automatically along with the window that contains it. If you do not wish a group to resize in this way, these restrictions are not necessary:

   - No active group can use the *layout* attribute.
   - Every group that is not an active group must have its CSS width and height attributes set to `100%`.

   The previous Zen page class example adheres to these rules. It also shows examples of using the *enclosingStyle* attribute, instead of a formal CSS style rule, to define the CSS width and height. For example:

   ```
   <group enclosingStyle="width:100%;height:100%;background:#ccffff;">
   ```

4. **onloadHandler** and **onlayoutHandler** methods in the Zen page class must call back into the Zen client side library module to ensure that any user adjustments to the window size force a recalculation of the active groups' geometry. The previous Zen page class example shows the correct calls, as follows:

```
ClientMethod onloadHandler() [ Language = javascript ]
  {
    ZLM.refreshLayout();
  }
ClientMethod onlayoutHandler() [ Language = javascript ]
  {
    ZLM.notifyResize(document.body);
  }
```

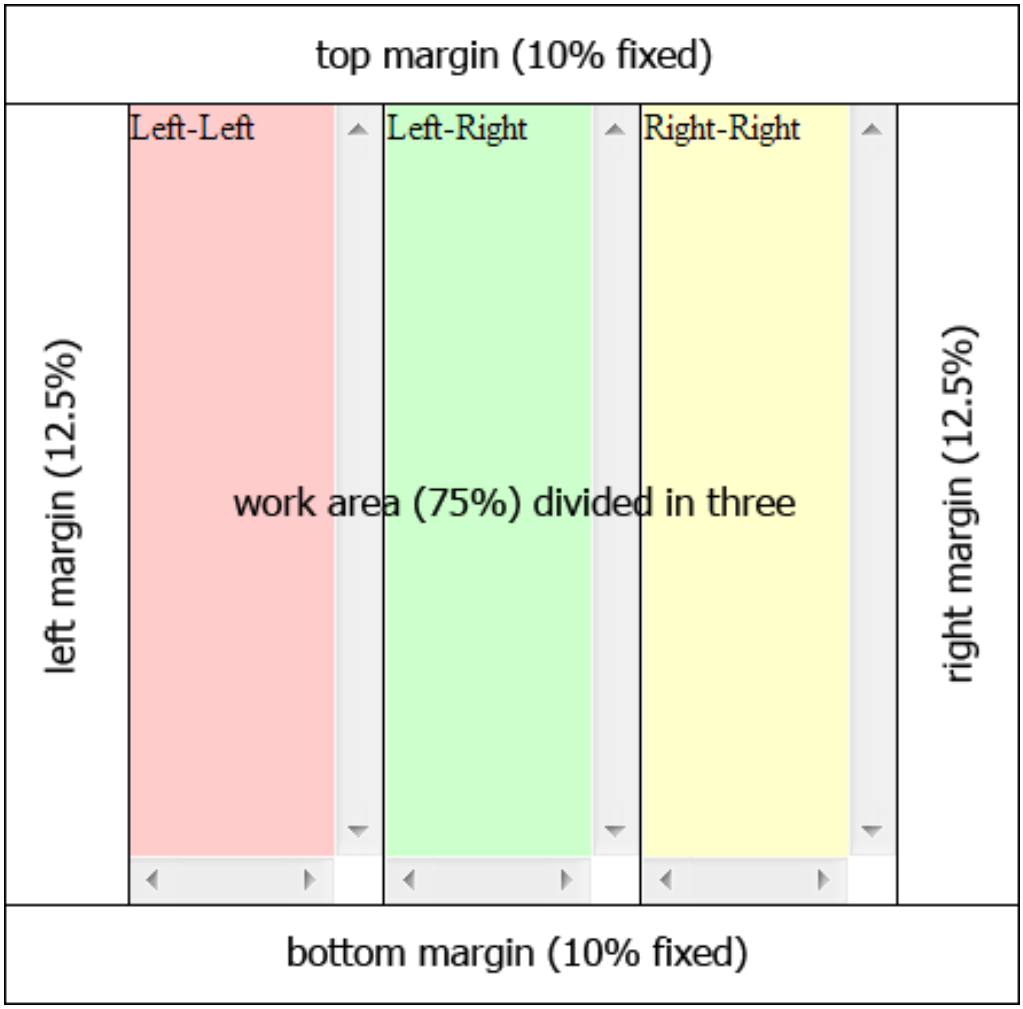
6.1.3 Calculating Percentages for Three-Way Splits

The following class code shows how to implement a page that requires areas of the screen to be split three ways, using the binary splits provided by <activeVGroup> and <activeHGroup>. This design has the following features:

- The screen has a 10% top margin, 10% bottom margin, 12.5% left and right margins and a work area in the middle. Thus, the main work area consumes 75% of the width and 80% of the height of the available pixels in the window.
- The main work area is divided into thirds. Each of these thirds has its own vertical scroll bar. Resizing the window resizes these areas automatically.
- Visible handle thicknesses shows how the screen is being cut up. Setting *handleThickness* to zero would make these handles disappear entirely.

The following figure shows a conceptual view of this design:

Figure 6–2: Three-Column Work Area Using Active Groups



You can create the layout by using active groups as shown in the following Zen page class:

Class Definition

```
Class MyApp.LayoutTest Extends %ZEN.Component.page
{

Parameter PAGENAME = "LayoutTest";

/// This Style block contains page-specific CSS style definitions.
XData Style
{
<style type="text/css">
#outerSplit {
    width:100%;
    height:100%;
  }
</style>
}

/// This XML block defines the contents of this page.
XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
<page xmlns="http://www.intersystems.com/zen" >
  <activeVGroup id="outerSplit" noResize="true"
                handleThickness="1" split="90%" >
    <activeVGroup id="topMarginSplit" noResize="true"
                  handleThickness="1" split="11%" >
      <!-- Top Margin -->
      <group id="Top"/>
      <activeHGroup id="rightMarginSplit" handleThickness="1"
                    noResize="true" split="88%" >
```

```
        <activeHGroup id="leftMarginSplit" handleThickness="1"
                      noResize="true" split="14%" >
          <!-- Left Margin -->
       <group id="Left"/>
          <group layout="none"
                 enclosingStyle="width:100%;height:100%;background:#ccccff;">
            <!-- Main Work Area -->
            <activeHGroup id="outerThreeway" noResize="true"
                          handleThickness="2" split="67%" >
              <activeHGroup id="innerThreeway" noResize="true"
                            handleThickness="2" split="50%" >
                <group id="leftLeft"
    enclosingStyle="width:100%;height:100%;overflow:scroll;background:#ffcccc;">
                  <label value="Left-Left"/>
                </group>
                <group id="leftRight"
    enclosingStyle="width:100%;height:100%;overflow:scroll;background:#ccffcc;">
                  <label value="Left-Right"/>
                </group>
              </activeHGroup>
              <group id="rightRight"
    enclosingStyle="width:100%;height:100%;overflow:scroll;background:#ffffcc;">
                <label value="Right-Right" />
              </group>
            </activeHGroup>
          </group>
        </activeHGroup>
        <!-- Right Margin -->
      </activeHGroup>
    </activeVGroup>
    <!-- Bottom Margin -->
  </activeVGroup>
</page>
}

ClientMethod onloadHandler() [ Language = javascript ]
  {
    ZLM.refreshLayout();
  }

ClientMethod onlayoutHandler() [ Language = javascript ]
  {
    ZLM.notifyResize(document.body);
  }
}
```

In this example, the binary nature of active group splits, combined with this design's requirement that all geometries be measured in percentages, requires the developer to pay close attention to the underlying constraints enforced by active groups. Within the main work area, the developer achieves an equal three-way split by nesting two horizontal splits. The outer split divides the total area at 67%, giving two-third of the horizontal pixels to the left child and one-third to the right. The inner split divides the left child at 50%, not 33%. Dividing the two-thirds area evenly in half yields the desired result of visually splitting the total area into equal thirds.

A similar technique correctly balances the left and right margins of the page relative to the central work area. The outer split divides the total area at 88%, giving the larger portion to the left child and creating a 12% right margin. The inner split divides the left child at 14%, not 12%. This produces a left margin that is of equal width to the right margin created by the outer split. The central work area is now approximately 75% of the window width and is correctly centered between the left and right margins.

6.2 Adding Objects Dynamically

In order to dynamically add a child object to an active group such as <desktop>, <corkboard>, or <snapGrid>, the active group needs to do client-side initialization of the child object.

The first step is to make sure that the entire DOM object is ready before calling the initialize code. Do this by calling the **refreshContents()** method with a parameter of 1 or true, to force a synchronous object creation.

Next re-initialize the active group and its children. You can do this with a call to **ZLM.initLayout()**, provided that the active group itself was statically defined as part of the base page.

You can call **ZLM.initLayout()** repeatedly as needed, but does take time to run, so if you know you are creating a number of drag groups in a row, it is best to call **refreshContents()** once after all the new children have been added, and call **ZLM.initLayout()** once after **refreshContents()** returns. You can add windows individually, but batching them when feasible cuts down on server chatter and client-side processing.

As an example, the following code shows the proper order for the calls to add an empty, resizable drag group from the client without a full page refresh.

```
XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
{
 <page xmlns="http://www.intersystems.com/zen">
  <hgroup>
   <button caption="add" onclick="zenPage.addClientSide();"/>
  </hgroup>
  <snapGrid id="snapGrid" cols="10" rows="10"
   enclosingStyle="height:500px;width:500px;background-color:silver">
  </snapGrid>
 </page>
}

ClientMethod addClientSide() [ Language = javascript ]
{
 var sg=zen("snapGrid");
 var dg=zenPage.createComponent("dragGroup");
 dg.homeCol=1;
 dg.homeRow=1;
 dg.rowSpan=2;
 dg.colSpan=2;
 dg.header="new";

 sg.addChild(dg);
 sg.refreshContents(1)
 ZLM.initLayout();
}
```

6.3 <activeHGroup> and <activeVGroup>

An <activeHGroup> is an active group that displays a two-part split pane with left and right partitions, optionally separated by a moveable adjustment bar. An <activeHGroup> can have only two child components. The first child defined in the <activeHGroup> appears in the left partition and the second child appears in the right partition. The “H” in the <activeHGroup> name means horizontal.

There is also an <activeVGroup> for the same functionality with vertical alignment. The split panes for <activeVGroup> are top and bottom rather than left and right.

Usually, both children of an <activeHGroup> or <activeVGroup> are groups. Each of these child groups defines the layout for its partition. The two children of an <activeHGroup> or <activeVGroup> may contain any number or type of Zen components, according to their usual allowances and restrictions, depending on which types of group they are: <hgroup>, <desktop>, <activeVGroup>, and so on.

<activeHGroup> and <activeVGroup> each have the following attributes.

Attribute	Description
Zen group attributes	<activeHGroup> and <activeVGroup> have the same style and layout attributes as other Zen groups, except that the *layout* attribute is not used for active groups. For descriptions, see “Group Layout and Style Attributes” in the “Zen Layout” chapter of *Using Zen*.

Attribute	Description
autoExpand	If defined, this property indicates that one of the two partitions is an automatic open, automatic close sidebar panel. When this is the case, the designated panel should expand when the mouse enters the bounds defined by the split between the partitions, and grow until it reaches the width given by the *autoExpand* value. The *autoExpand* value is always interpreted in pixels. A positive value designates the left (top) partition for automatic expansion. A negative value designates the right (bottom) partition. The default value (null) disables the feature for this group.
handlePattern	String indicating a file name for an image to use to paint the dragable partition handle. For <activeHGroup>, this image should be at least as wide as the handle thickness and is repeated vertically along the length of the handle. Correspondingly, for <activeVGroup>, this image should be at least as tall as the handle height and is repeated horizontally along the length of the handle. The default is a PNG image that produces a gray gradient effect.
handleThickness	Width in pixels of the adjustment handle, which is displayed along the dividing line between the two partitions. Due to the dynamic nature of active groups, the width of this bar cannot be set using CSS and must be specified via the *handleThickness* property. The default is "7" which produces a drag handle of one-eighth inch (2 mm) on most screens.
noResize	If true, the user is not allowed to resize the partitions. If false, the user can adjust partition size by dragging the adjustment handle with the mouse. The default is false. If resizing is enabled (*noResize* is false) the mouse pointer changes to an east-west (or north-south) resize cursor when the mouse is in a potential drag position.
onresizeBottom	Specifies a handler for the onresizeBottom event. This event fires when the user resizes the bottom panel of this <activeVGroup>.
onresizeLeft	Specifies a handler for the onresizeLeft event. This event fires when the user resizes the left panel of this <activeHGroup>.
onresizeRight	Specifies a handler for the onresizeRight event. This event fires when the user resizes the right panel of this <activeHGroup>.
onresizeTop	Specifies a handler for the onresizeTop event. This event fires when the user resizes the top panel of this <activeVGroup>.
soundFX	String indicating a file name for a sound file to be played when an automatic expansion group grows or shrinks. If not defined or null (a blank string) no sound accompanies the animation. The default is null.

Attribute	Description
split	A string that defines the division between the two panes. Zen interprets the *split* value as follows: • A value that ends in % (the percent sign) is interpreted as a proportional division between the left and right (or top and bottom) panes and is recalculated whenever the base container is resized. • A positive value without % is interpreted as a fixed width for the left (or top) partition. In this case, any adjustments to the total width of the base container affect the right (or bottom) partition only. • A negative value (which cannot contain a % sign) is interpreted as a fixed width for the right (or bottom) partition. In this case, any adjustments to the total width of the base container affect the left (or top) partition only. The default *split* value is this string: `50%` Manual adjustment of the panes changes the size of the fixed width or percentage. The new sizes or ratios are respected when the base container is resized.

<activeHGroup> and <activeVGroup> divide the screen as specified by the *split* property based on the size of the window at the time the page is loaded, and allow the user to readjust this layout as desired with the adjustment handles. You can reset the size of the panes programmatically, including changing whether the split is calculated as a fixed size or a ratio. If the value of the *split* property is a string ending in a percent sign, such as “75%,” the split is calculated as a ratio. If the value of *split* is a string ending in px, such as “75px,” the split is calculated as a fixed width. If the value of *split* is a number, such as `50%`, the split is resized using the specified number and retaining whatever units are in use at the time. For example, the following code fragment creates an <activeVGroup> with *split* set to the default value of `50%`:

XML

```
<activeVGroup id="activeVGroup" handleThickness="5">
 <vgroup>
  <hgroup>
   <button caption="Change To Pixels" valign="top"
    onclick="zenPage.changeSize('pixels');" />
   <button caption="Change To Percent" valign="top"
    onclick="zenPage.changeSize('percent');" />
   <button caption="Change Size Same Units" valign="top"
    onclick="zenPage.changeSize('same');" />
  </hgroup>
 </vgroup>
</activeVGroup>
```

The *onclick* event handler for the buttons changes the size and units of the split, as follows:

Class Member

```
ClientMethod changeSize(mode) [ Language = javascript ]
{
  var avgr = zenPage.getComponentById('activeVGroup');
    switch (mode)
    {
      case 'pixels':
        avgr.setProperty('split',"75px");
        break;
      case 'percent':
        avgr.setProperty('split',"75%");
        break;
      case 'same':
        avgr.setProperty('split',30);
        break;
    }
}
```

If you want the groups to automatically recalculate their layouts in response to a resize of the base window itself, you need to add the following call inside the optional **onlayoutHandler** JavaScript callback method for the Zen page:

```
ZLM.notifyResize(document.body);
```

When this call is in place, the **notifyResize** function is called automatically both when the page first loads and in response to window resizing at the browser level. Placing **notifyResize** in the **onlayoutHandler** callback allows you to place other "resizing reactions" in the **onlayoutHandler** callback, if you desire further processing.

For more about **onlayoutHandler**, see the section "Zen Layout Handler."

6.4 <corkboard>

A <corkboard> may contain one or more <dragGroup> components. A <corkboard> is an active group whose child components may visually overlap. As soon as the user clicks on a <dragGroup> in a <corkboard>, that <dragGroup> comes to the foreground. The user can subsequently drag the <dragGroup> and drop it in a new position in the <corkboard>. The dragged component overlays everything else inside the <corkboard>. No other component, besides the dragged component, is moved out of its current position.

The user cannot move components into or out of the <corkboard>. Direct manipulation can only occur *within* the <corkboard> container.

The immediate children of a <corkboard> component *must* be <dragGroup> components, but there are no restrictions placed on the contents of the <dragGroup> components themselves. A <dragGroup> may contain any component that is valid in a group.

<corkboard> has the following attributes.

Attribute	Description
Zen group attributes	<corkboard> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.

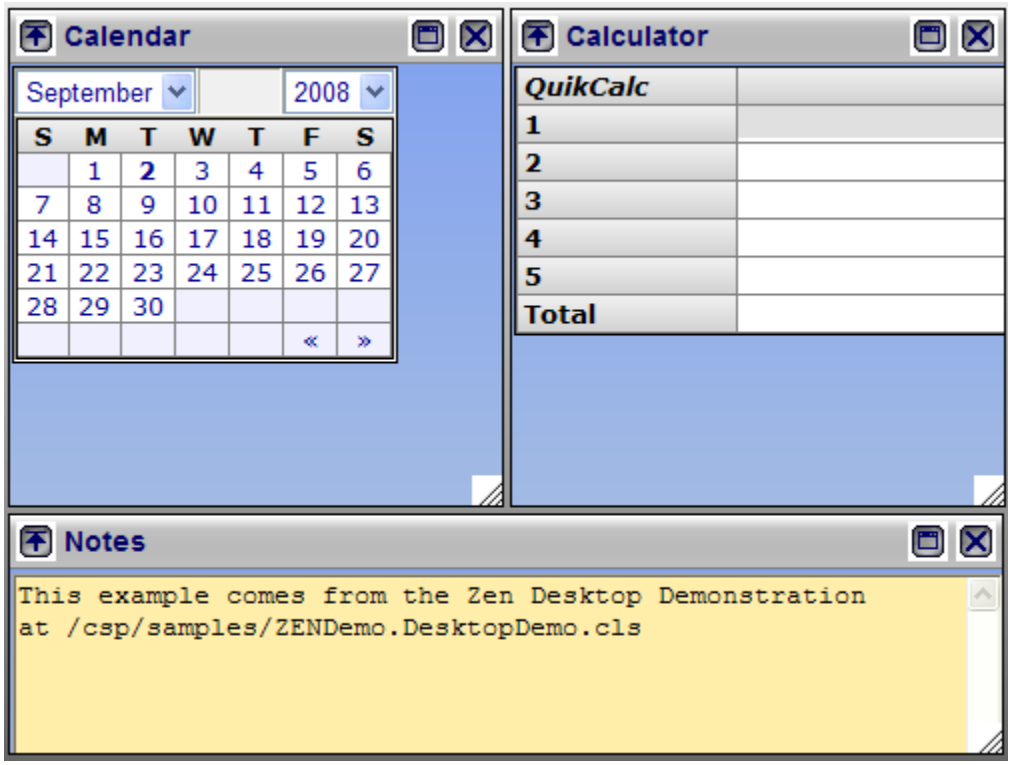
6.5 <desktop>

A <desktop> may contain one or more <dragGroup> components. A <desktop> is an active group that "tiles" its child components so that each one is fully visible and none of them overlap. As soon as the user clicks on a <dragGroup> in a <desktop>, that <dragGroup> comes to the foreground. The user can subsequently drag the <dragGroup> and drop it in a new position in the <desktop>. When the user drags and drops a <dragGroup> component into a new position, the dragged component moves the other <dragGroup> components in the <desktop> out of its way. All <dragGroup> components change position after the dragged component is dropped.

The user cannot move components into or out of the <desktop>. Direct manipulation can only occur *within* the <desktop> container.

The immediate children of a <desktop> component *must* be <dragGroup> components, but there are no restrictions placed on the contents of the <dragGroup> components themselves. A <dragGroup> may contain any component that is valid in a group.

<desktop> has the following attributes.



The underlying geometric model for the <desktop> is a simple one based on rows and columns. For this reason the <desktop> component supports row and column style settings that can enforce row and column alignment constraints on the <dragGroup> components that it contains.

The arrangement of <dragGroup> components within the <desktop> varies, but in general this arrangement is biased in favor of "row collapse." This means that if a <dragGroup> is removed from a given row, any other groups in the same row move to the left, collapsing the length of the row and possibly creating blank space at the extreme right end of the row. This convention maximizes the use of the visible portions of the window and minimizes the need for horizontal scrolling.

You can suggest an initial layout of the <dragGroup> components within the <desktop> by supplying CSS styles such as `valign:top` and `align:left` for these components. You can also set the CSS style properties `top` and `left` to explicit pixel values; for example:

XML

```
<dragGroup enclosingStyle="top:150px; left:233px;" >
    <!-- contents of drag group here -->
</dragGroup>
```

When rendering the <desktop> component, Zen makes every effort to abide by such suggestions within the other constraints that are currently active. However, row and column styles, automatic row collapse, and the prohibition against overlapping <dragGroup> components make it impossible to guarantee that the placement suggested via CSS is respected exactly.

The <desktop> component has the following attributes.

Attribute	Description
Zen group attributes	In addition to its capabilities as an active group, <desktop> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
colStyle	The <desktop> can enforce sizing and alignment constraints on the <dragGroup> components within its columns. You can choose a column configuration by assigning a value to the *colStyle* attribute as described in the section "<desktop> Column Style" following this table.
rowStyle	The <desktop> can enforce sizing and alignment constraints on the <dragGroup> components within its rows. You can choose a row configuration by assigning a value to the *rowStyle* attribute as described in the section "<desktop> Row Style" following this table.

It is possible to query, save, and restore the <desktop> state. The <desktop> component provides JavaScript methods that allow you to save and restore a particular user's choice of geometry for the <desktop> — that is, how the user sized and placed components within the <desktop> during a previous session with the Zen page. You can work with this information as follows:

getState

layout=`getState()`

The **getState** method returns a string that records the <desktop> state. The format for this string is an internal encoding that you do not need to understand in order to use this method. Simply store the string, in any way desired, to retrieve it for later use with **restoreState**.

restoreState

`restoreState(`*layout*`)`

Inputs a string that tells the <desktop> which geometry to use when it displays. *layout* must be a string that was previously acquired using **getState**. The **restoreState** method does not actually open any windows. It takes whatever windows happen to be open, and restores them to where they were when the *layout* string was saved.

6.5.1 <desktop> Row Style

The <desktop> can enforce sizing and alignment constraints on the <dragGroup> components within its rows. You can choose a configuration by assigning a value to the *rowStyle* attribute. Valid values are:

- FILL_ROW — All groups within a row have the height of the tallest group in the row. By default the top and bottom edges of all the sub-windows align. Height is allowed to vary from one row to the next.
- ALIGN_ROW_TOP — (the default) The top edge of all <dragGroup> components within a row align. The height of individual <dragGroup> components within the row is allowed to vary. The row spacing is driven by the tallest group within the row.
- ALIGN_ROW_CENTER — All <dragGroup> components are centered vertically within their respective rows. The height of individual <dragGroup> components is allowed to vary. The row spacing is driven by the tallest group within the row.
- ALIGN_ROW_BOTTOM — The bottom edge of all <dragGroup> components within a row align. The height of individual <dragGroup> components within the row is allowed to vary. The row spacing is driven by the tallest group within the row.

- FILL_UNIFORM — All <dragGroup> components within the desktop take on the height of the tallest <dragGroup>. This results in all rows being uniformly spaced vertically and all <dragGroup> components being both top and bottom aligned.
- ALIGN_UNIFORM_TOP — All rows within the desktop take on uniform spacing dictated by the height of the tallest group within the component. Within each row, the top edges of individual <dragGroup> components align. The height of individual <dragGroup> components is allowed to vary.
- ALIGN_UNIFORM_CENTER — All rows within the desktop take on uniform spacing dictated by the height of the tallest group within the component. Within each row, individual <dragGroup> components are centered vertically. The height of individual <dragGroup> components is allowed to vary.
- ALIGN_UNIFORM_BOTTOM — All rows within the desktop take on uniform spacing dictated by the height of the tallest group within the component. Within each row, the bottom edges of individual <dragGroup> components align. The height of individual <dragGroup> components is allowed to vary.

6.5.2 <desktop> Column Style

In general, the <desktop> geometry conventions are biased towards rows. In fact, by default the <desktop> does not recognize the existence of columns.

However, assigning a value to the *colStyle* attribute alters this behavior, so that the <desktop> counts all the first elements of the rows as column one, all the second elements as column two, and so forth. Unlike rows, which repack themselves when they are removed or added, columns are allowed to have embedded gaps in which a short row does not reach a given column.

You can choose a column style, and force <desktop> to recognize columns, by assigning a value to the *colStyle* attribute. If you do not, the concept of columns is ignored and only row-based constraints apply.

Valid values for *colStyle* are:

- FILL_COLUMN — All <dragGroup> components within a given column take on the width of the widest group within the column. All <dragGroup> components become both left and right aligned. The spacing of the columns is allowed to vary from one column to the next
- ALIGN_COLUMN_LEFT — The width of the column is dictated by the width of the widest group within that column but the width of individual sub- windows is allowed to vary. All <dragGroup> components are left aligned within their columns. The spacing of the columns is allowed to vary from one column to the next
- ALIGN_COLUMN_CENTER — The width of the column is dictated by the width of the widest group within that column, but the width of individual <dragGroup> components is allowed to vary. All <dragGroup> components are centered within their columns. The spacing of the columns is allowed to vary from one column to the next
- ALIGN_COLUMN_RIGHT — The width of the column is dictated by the width of the widest group within that column, but the width of individual <dragGroup> components is allowed to vary. All <dragGroup> components are right aligned within their columns. The spacing of the columns is allowed to vary from one column to the next
- FILL_WIDTH — The total width of the longest row dictates the layout bounds for the entire <desktop>. The widths of <dragGroup> components within shorter rows are scaled up proportionately to ensure that right edge of the last group in each row is aligned with that of every other row in the <desktop>.
- ALIGN_WIDTH_LEFT — Similar to FILL_WIDTH. The <dragGroup> components within rows are horizontally spaced based on the width of the longest row. The widths of individual <dragGroup> components are not padded, creating the appearance of random spacing within a row.
- ALIGN_WIDTH_CENTER — Similar to FILL_WIDTH. The <dragGroup> components within rows are horizontally spaced based on the width of the longest row. The widths of individual <dragGroup> components are not padded, and the <dragGroup> components are centered within the revised spacing bounds.

- ALIGN_WIDTH_RIGHT — Similar to FILL_WIDTH. The <dragGroup> components within rows are horizontally spaced based on the width of the longest row. The widths of individual <dragGroup> components are not padded, and the <dragGroup> components are right aligned within the revised spacing bounds.
- FILL_UNIFORM — All columns take on the width and spacing dictated by the widest <dragGroup> within the <desktop>. All <dragGroup> components are given uniform width, and are automatically left and right aligned
- ALIGN_UNIFORM_LEFT — Similar to FILL_UNIFORM, but all <dragGroup> components within a column are left aligned.
- ALIGN_UNIFORM_CENTER — Similar to FILL_UNIFORM, but all <dragGroup> components within a column are centered within the column spacing.
- ALIGN_UNIFORM_RIGHT — Similar to FILL_UNIFORM, but all <dragGroup> components within a column are right aligned.

6.6 <snapGrid>

A <snapGrid> is an active group which can contain one or more <dragGroup> components, organized so that each one is aligned with a grid. The number of rows and columns specified defines the underlying grid. The resulting grid is a normalized space, meaning that a four column layout results in the width of each column being 25% of the total width.

You can define different layouts for portrait and landscape page orientation, such that the number of columns and rows changes if the geometry of the <snapGrid> becomes taller than it is wide (or vise versa). This is particularly useful for adapting layouts on devices such as tablets and mobile phones.

The immediate children of a <snapGrid> component *must* be <dragGroup> components, but there are no restrictions placed on the contents of the <dragGroup> components themselves. A <dragGroup> may contain any component that is a valid child of a group.

Child components may visually overlap. A <dragGroup> in a <snapGrid> comes to the foreground when the user clicks on it. If the appropriate user gestures are enabled, the user can drag and drop the <dragGroup>. When dropped, a <dragGroup> snaps to the closest grid lines. The grid itself does not scroll, however the <dragGroup> components in the grid may.

The user cannot move components into or out of the <snapGrid>. Direct manipulation can only occur *within* the <snapGrid> container.

<snapGrid> has the following attributes.

Attribute	Description
Zen group attributes	In addition to its capabilities as an active group, <snapGrid> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
cols	Specifies the number of columns used for both portrait and landscape layouts. The default value is 3.
colsLandscape	Specifies the number of columns used when the rendered width of the <snapGrid> is greater than or equal to its height.
colsPortrait	Specifies the number of columns used when the rendered width of the <snapGrid> is less than its height.
rows	Specifies the number of rows used for both portrait and landscape layouts. The default value is 2.
rowsLandscape	Specifies the number of rows used when the rendered width of the <snapGrid> is greater than or equal to its height.
rowsPortrait	Specifies the number of rows when the rendered width of the <snapGrid> is less than its height.

6.6.1 Dynamic and Static Layout

The ability to resize and drag and drop <dragGroup> components in a <snapGrid> is primarily controlled by the <dragGroup>. The attributes *moveEnabled* and *resizeEnabled* specify whether you can move or resize the group. <dragGroup> also relies on the header and the resize button to provide affordances for user drag and resize gestures. Without them, the drag group cannot be moved or resized. You can use the <snapGrid> method **broadcast** to control various aspects of appearance and behavior for all <dragGroup> components in the grid. The method **restyleHeaderStyles** in the class ZENDemo.SnapGridDemo in the SAMPLES namespace illustrates this use of **broadcast**.

broadcast sends a signal to each of the active drag group components in the snap grid. Valid signals include:

- resize – call the onresize handler for each of the drag groups.
- disableResize – remove the resize button from the lower right corners of the drag groups.
- disableMaxToggle – redefine the drag group header to be a drag handle rather than a maximize button.
- enableMaxToggle – redefine the drag group header to be a maximize button rather than a drag handle.
- enableResize – restore the resize button to the lower right corners of the drag groups.
- processAppMessage – simply pass the value given to the <dragGroup> children. Subclasses of <dragGroup> can override the default processAppMessage() method to extend the broadcast facility to address application specific signals. The mechanism only passes a single value argument, but this value can take the form of a JSON string or an arbitrary JavaScript object so multiple discrete data can be packaged into a single broadcast value.
- removeDragHeader – completely hide the drag handle and resize the window to use the recovered space.
- restoreDragHeader – restore the drag handle and resize the window to account for the newly used space.
- setBodyStyle– given a JSON representation of a style object in value, add the given rules to the existing style of the drag body.
- setHeaderStyle – given a JSON representation of a style object in value, add the given rules to the existing style of the drag header.
- setHeaderLayout – set the headerLayout property of all child drag groups to the given value.

Note that when you run ZENDemo.SnapGridDemo, it comes up with headers and resize buttons visible, and you can move and resize the drag groups. The **Remove Headers** button removes these affordances, and as a result, the grid layout is static. The **Restyle Headers** button restores the headers functioning as maximize buttons.

6.6.2 Orientation-specific Layout

The <snapGrid> component can adjust the layout of rows and columns in response to the geometry of the window it is being viewed in. The properties *rowsPortrait* and *colsPortrait* specify the number of rows and columns for portrait mode, and *rowsLandscape* and *colsLandscape* for landscape mode. If you set only one of these two sets of properties, Zen uses the default row and column values for the unspecified set.

This feature is useful when creating applications to be viewed on a device such as a cell phone or tables, that changes the display orientation when the orientation of the device changes. The following example codes a page that responds to changes in window geometry when viewed in a browser. CSS sets the height and width of the <snapGrid> to 100%, and the page layout must be set to "none" as described in the section "Active Groups that Resize Proportionally."

```
Class demo.SnapGridLP Extends %ZEN.Component.page
{
 /// This Style block contains page-specific CSS style definitions.
 XData Style
 {
  <style type="text/css">
   #snapGrid {
    width: 100%;
    height: 100%;
    background-color: silver;
   }
  </style>
}
 /// This XML block defines the contents of this page.
 XData Contents [ XMLNamespace = "http://www.intersystems.com/zen" ]
 {
  <page xmlns="http://www.intersystems.com/zen"
   layout="none">
   <snapGrid id="snapGrid"
    colsLandscape="10"
    colsPortrait="5"
    rowsLandscape="5"
    rowsPortrait="10"
   >
    <dragGroup id="dg1" onclick=""
     moveEnabled="true" resizeEnabled="true"
     headerLayout="C" header="1"
     homeCol="1" homeRow="1"
     colSpan="1" rowSpan="1">
    </dragGroup>
   </snapGrid>
  </page>
 }
}
```

6.7 <dragGroup>

A <dragGroup> is the only Zen group component that can be the direct child of a <desktop>, <corkboard>, or <snapGrid>. The <dragGroup> itself can contain any component you can put in a group.

A <dragGroup> can display a header bar, which can have a text label and various buttons. A <dragGroup> can be resized, maximized, minimized, closed, and restored by clicking its buttons and by dragging the header or the resize icon in the lower right corner. In many ways the <dragGroup> is similar to a desktop window such as you would typically see on a Macintosh or PC desktop.

<dragGroup> has the following attributes.

Attribute	Description
Zen group attributes	<dragGroup> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
centerHeader	If set, this boolean flag indicates that the title section of the header should be centered over the <dragGroup>.
colSpan	If the <dragGroup> is a child of a <snapGroup>, this specifies the initial width of the <dragGroup> in columns of the <snapGroup>. See Dynamic and Static Layout for more information on resizing a <dragGroup> in a <snapGroup>.
header	Title to display in the header bar that displays across the top of this <dragGroup>. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. This makes it easy to localize its 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 *invalidMessage* property from client-side or server-side code.
headerLayout	The header layout pattern determines the order in which controls are added to the header section of the <dragGroup> window frame. This is encoded as a five character string consisting of the following tokens : • 'I' represents the Iconify button • 'T' represents the Title section with both application logo and header caption • 'F' represents the Full screen button • 'C' represents the Close button • 'U' represents the User content div (if desired) This string allows you restructure the location of the controls with a single call and can be useful if you are trying to match the look and feel of a given operating system. For example, MS-Windows systems follow the pattern of 'TIFC' whereas macOS adopts the standard of CIFT with the title centered (see *centerHeader*). The Title section represent something of a breakpoint in the header layout. Everything prior to the header floats from the left end of the header; everything after the header, floats from the right end.
homeCol	If the <dragGroup> is a child of a <snapGroup>, this specifies the initial column where the top of the <dragGroup> is positioned. See Dynamic and Static Layout for more information on repositioning a <dragGroup> in a <snapGroup>.
homeRow	If the <dragGroup> is a child of a <snapGroup>, this specifies the initial row where the left edge of the <dragGroup> is positioned. See Dynamic and Static Layout for more information on repositioning a <dragGroup> in a <snapGroup>.

Attribute	Description
imageClose	URI of the image to display on the button that the user clicks to close this <dragGroup>. If you do not specify *imageClose*, Zen uses its default:  If *imageClose* is the relative pathname of a file, it is understood to be relative to the CSP directory in the Caché installation directory. Typically this path identifies the images subdirectory for your Zen application. For example, the following value: `<dragGroup imageClose="myApp/images/myClose.png"/>` finds image files in the directory: *install-dir*`/CSP/`*myNamespace*`/myApp/images/myClose.png` This holds true for any image attribute of <dragGroup>.
imageCloseHover	The URI of the image to display for the close group button when the mouse hovers over the icon.
imageCloseWidth	The width (in pixels) of the space allocated for the images for the close group button. The default images are 16 by 16 pixels. You cannot specify a different image height.
imageContract	URI of the image to display on the button that the user clicks to contract this <dragGroup>. If you do not specify *imageContract*, Zen uses its default:  . See *imageClose*.
imageContractHover	The URI of the image to display for the contract group button when the mouse hovers over the icon.
imageContractWidth	The width (in pixels) of the space allocated for the images for the contract group button.
imageExpand	URI of the image to display on the button that the user clicks to expand this <dragGroup>. If you do not specify *imageExpand*, Zen uses its default: 
imageExpandHover	The URI of the image to display for the expand group button when the mouse hovers over the icon.
imageExpandWidth	The width (in pixels) of the space allocated for the images for the expand group button.
imageMinimize	URI of the image to display on the button that the user clicks to minimize this <dragGroup>. If you do not specify *imageMinimize*, Zen uses its default: 
imageMinimizeHover	The URI of the image to display for the minimize group button when the mouse hovers over the icon.
imageMinimizeWidth	The width (in pixels) of the space allocated for the images for the minimize group button.

Attribute	Description
imageResize	URI of the image to display on the button that the user clicks to resize this <dragGroup>. If you do not specify *imageResize*, Zen uses its default: 
imageResizeSize	Size of the square to reserve for the resize icon. This is an integer value, in pixels. The default value is 16, which indicates a 16x16 pixel square.
minWidth	This sets the minimum width (in pixels) for a dragGroup when the user is resizing it interactively. The value of *minWidth* has no bearing on programmatic setting of the dragGroup's size. If used within a snapGrid context, the actual minimum width is set to the next highest even multiple of the snapGrid column width. The property is an integer with the implied units of pixels. It can be set either statically in the XData block or at run time with a call to **setProperty()** on the object. The default value is 96.
moveEnabled	If true, this flag allows the user to reposition the <dragGroup> by drag and drop gestures within the containing active group area. The default value is true. *moveEnabled* has the underlying data type %ZEN.Datatype.boolean. See Zen Attribute Data Types.
onclosepending	The *onclosepending* event handler: This event is fired when the user clicks the close button of this <dragGroup>. Unlike native browser windows, it is possible to prevent the window closure by calling **abortClose()** in response to this event.
onresize	The *onresize* event handler: This event is fired when the user resizes the <dragGroup>.
onwindowdrop	The *onwindowdrop* event handler: This event is fired when the user releases the title bar of this <dragGroup> after a drag and drop gesture.
onwindowgrab	The *onwindowgrab* event handler: This event is fired when the user grabs this <dragGroup> by the title bar for a drag and drop gesture.
resizeEnabled	If true, this flag allows the user to resize this <dragGroup>. The default value is true. *resizeEnabled* has the underlying data type %ZEN.Datatype.boolean. See Zen Attribute Data Types.
rowSpan	If the <dragGroup> is a child of a <snapGroup>, this specifies the initial height of the <dragGroup> in columns of the <snapGroup>. See Dynamic and Static Layout for more information on resizing a <dragGroup> in a <snapGroup>.

7

Client Side Menu Components

This chapter describes built-in Zen components that are designed to make it easy to render a more "native application" look and feel for Zen pages through the use of a client side menu system. These menu components provide simple mechanisms for JavaScript post-processing on the client side, so that their appearance and layout can respond to user actions on the client side.

Note: The style for more traditional menu components is defined with CSS and HTML, without the option for subsequent post-processing in JavaScript. For full information about those components, see the "Navigation Components" chapter of *Using Zen Components*.

7.1 Building Client Side Menus

The original <menu> component offers a simple HTML label in a box with all of its (limited) rendering done at the server with a little help from CSS preferences. It is perfectly serviceable as a menu, but its appearance has very little flexibility.

In contrast, the <csMenuBar> defers its rendering until it has actually been loaded by the client and has access to more information about the user's machine (browser, operating system, available fonts, etc). The <csMenuBar> makes heavy use of JavaScript to do things that are difficult (or in some cases impossible) to do with CSS alone. It can render a menu that looks and acts more like a user might find in applications such as Microsoft Outlook or Word. This menu includes features such as icons, keyboard shortcuts, scrolling of long menus without scrollbars, and so on.

In general, the cs prefixed widgets such as <csMenuBar> form one set of tools, while the traditional menu widgets described in the "Navigation Components" chapter of *Using Zen Components* form another. These sets are not interchangeable. For example, a <csMenuBar> consists of <csMenuBarItem>, <contextMenu>, <csMenuItem>, and <csMenuSeparator> elements. A mixed specification in which <csMenuItem> elements are separated by <menuSeparator> elements causes the menu subsystem to break.

It is important to note that there is inconsistency in the naming conventions. Not all of the client side widget names start with cs. The cs prefix is used only when the newer, client side widget name conflicts with the older, original widget names. Unfortunately this means that <contextMenu>, the widget at the heart of the client side menu subsystem, does not use the cs prefix, but every element around it does.

The following is a summary of the nesting structure for the client side menu elements:

XML

```
<csMenuBar>
    <csMenuBarItem>
        <contextMenu>
            <csMenuItem />
            <csMenuItem />
            <csMenuSeparator />
            <csMenuItem />
            <csMenuItem>
                <contextMenu>
                    <csMenuItem />
                    <csMenuItem />
                </contextMenu>
            </csMenuItem>
        </contextMenu>
    </csMenuBarItem>
</csMenuBar>
```

The next several sections describe how to work with client side menu components to provide:

- Drop down menus
- Menu picks or choices
- Popup or context menus
- Cascading menus
- Button bars
- Icons

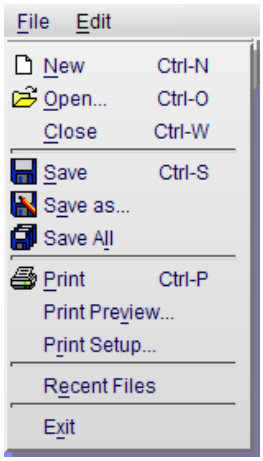
For a syntax guide to specific components, see the later sections in this chapter:

- <csMenuBar>
- <csMenuBarItem>
- <contextMenu>
- <csMenuItem>
- <csMenuSeparator>
- <buttonBar>
- <buttonBarItem>

7.1.1 Drop Down Menus

The following sample <csMenuBar> element defines a horizontal menu bar with two picks: File and Edit. Each of these picks displays a different <contextMenu>. The <csMenuItem> components in this example illustrate a variety of options for setting associated icons, alternate captions, keyboard shortcuts, and so on. The following figure illustrates the File pick from this menu.

Figure 7–1: File Drop Down Menu with Icons and Keyboard Shortcuts



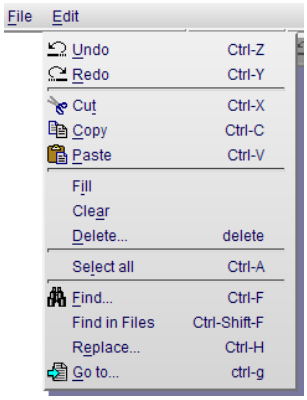
XML

```
<csMenuBar id="mainMenu" width="100%">
  <csMenuBarItem caption="File" contextKey="f" >
    <contextMenu id="fileMenu" >
      <csMenuItem icon="images/file.png" caption="New" key="Ctrl-N"
                  contextKey="n" onclick="alert('New requested');"  />
      <csMenuItem icon="/csp/broker/images/open.png" caption="Open..."
                  key="Ctrl-O" contextKey="o"
                  onclick="alert('Open requested');"  />
      <csMenuItem caption="Close" key="Ctrl-W"
                  contextKey="c" onclick="alert('Open requested');"  />
      <csMenuSeparator />
      <csMenuItem id="savePick" icon="/csp/broker/images/save.png"
                  iconDisabled="/csp/broker/user/images/save_g.png"
                  caption="Save" key="Ctrl-S" contextKey="s"
                  onclick="alert('Save requested');"  />
      <csMenuItem icon="/csp/broker/images/saveas.png" caption="Save as..."
                  contextKey="a" />
      <csMenuItem icon="/csp/broker/images/saveall.png" caption="Save All"
                  contextKey="l" />
      <csMenuSeparator />
      <csMenuItem icon="/csp/broker/images/print.png" caption="Print"
                  key="Ctrl-P" contextKey="p"
                  onclick="alert('Print requested');" />
      <csMenuItem caption="Print Preview..."  contextKey="v" />
      <csMenuItem caption="Print Setup..."  contextKey="r" />
      <csMenuSeparator />
      <csMenuItem caption="Recent Files"  contextKey="e" />
      <csMenuSeparator />
      <csMenuItem caption="Exit" contextKey="x" />
    </contextMenu>
  </csMenuBarItem>
  <csMenuBarItem caption="Edit" contextKey="e">
    <contextMenu id="editMenu" >
      <csMenuItem id="undoPick" icon="/csp/broker/images/undo.png"
                  caption="Undo" key="Ctrl-Z" altCaption="Can't undo"
                  contextKey="u" onclick="alert('undo');" />
      <csMenuItem id="redoPick" icon="/csp/broker/images/redo.png"
                  caption="Redo" key="Ctrl-Y" altCaption="Can't redo"
                  contextKey="r" onclick="alert('redo');" />
      <csMenuSeparator />
      <csMenuItem id="cutPick" icon="/csp/broker/images/cut.png"
                  caption="Cut" key="Ctrl-X" contextKey="t"
                  onclick="alert('cut');" />
      <csMenuItem id="copyPick" icon="/csp/broker/images/copy.png"
                  caption="Copy" key="Ctrl-C" contextKey="c"
                  onclick="alert('copy');"  />
      <csMenuItem id="pastePick" icon="/csp/broker/images/paste.png"
                  caption="Paste" key="Ctrl-V" contextKey="p"
                  onclick="alert('paste');"  />
      <csMenuSeparator />
      <csMenuItem caption="Fill" contextKey="i" onclick="alert('fill');" />
      <csMenuItem caption="Clear" contextKey="a" onclick="alert('clear');" />
      <csMenuItem caption="Delete..." key="delete"
                  contextKey="d" onclick="alert('delete');"  />
      <csMenuSeparator />
      <csMenuItem caption="Select all" key="Ctrl-A"
```

```
                contextKey="l" onclick="alert('select all');" />
      <csMenuSeparator />
      <csMenuItem icon="/csp/broker/images/find.png" caption="Find..."
                  key="Ctrl-F" contextKey="f" onclick="alert('find');" />
      <csMenuItem caption="Find in Files" key="Ctrl-Shift-F"
                  onclick="alert('find next');" />
      <csMenuItem caption="Replace..." key="Ctrl-H" contextKey="e"
                  onclick="alert('replace');" />
      <csMenuItem icon="/csp/broker/images/goto.png" caption="Go to..."
                  key="ctrl-g" contextKey="g" onclick="alert('goto');" />
    </contextMenu>
  </csMenuBarItem>
</csMenuBar>
```

The following figure illustrates the Edit pick from this menu.

Figure 7–2: Edit Drop Down Menu with Icons and Keyboard Shortcuts



7.1.2 Menu Picks or Choices

Zen client side menus support the following types of menu pick:

Type of Pick	User Action	Result
Regular menu	Click	Zen invokes the callback specified by the *onclick* attribute. You can define a regular menu pick with just a *caption* and an *onclick* callback, but you can also add an *icon*, or a keyboard shortcut using *key* or *contextKey*. For example: For some images that you can use in menu picks, see the section "Icons."
Cascading menu	Roll over	Zen displays a submenu. You can cascade menus by nesting <contextMenu> elements within <csMenuItem> elements. See the examples in the sections "Building Client Side Menus" and "Popup or Context Menus."

Type of Pick	User Action	Result
Alternating caption	Click	Zen invokes the callback specified by the *onclick* attribute. The *toggleState* property flips its value (from 0 to 1 or from 1 to 0) when the user clicks the menu pick. The caption label changes automatically, as in “Show” vs. “Hide.” For an alternating caption pick, define both a *caption* and an alternate caption (*altCaption*) as shown in the following example: Optionally, you can also specify an *icon* and *altIcon* so that the icon can be toggled along with the caption. A *toggleState* of 0 displays the *caption* and *icon*; 1 displays the *altCaption* and *altIcon*. The *toggleState* changes only in response to user actions.

7.1.3 Popup or Context Menus

You can use client-side menu components to define a popup or context menu for a region. To accomplish this, wrap a <contextMenu> definition inside a group of some kind. The wrapper component can be nearly anything, but is most often a reasonably significant portion of the page: a <vgroup>, <hgroup>, <form>, etc. Define the <contextMenu> as a child of the wrapper component, and a sibling of other elements within the wrapper. You can optionally specify exact bounds, as for the <group> shown in the next example, but this is not required.

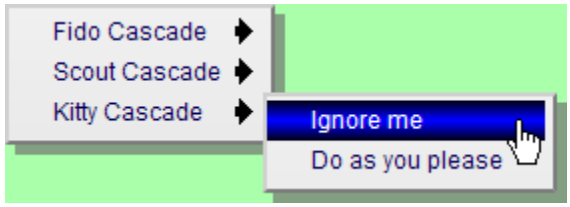
Subsequently, when the user right-clicks the mouse anywhere within the wrapper component, Zen displays and activates the <contextMenu> child component, rather than the browser’s default right-click menu.

XML

```
<group width="400" height="800" enclosingStyle="background:#aaffaa;" >
  <contextMenu id="rightPopUp" >
    <csMenuItem caption="Fido Cascade" >
      <contextMenu >
        <csMenuItem caption="Sit" />
        <csMenuItem caption="Stay" />
        <csMenuItem caption="Play dead" />
      </contextMenu>
    </csMenuItem>
    <csMenuItem caption="Scout Cascade" >
      <contextMenu >
        <csMenuItem caption="Whoa" />
        <csMenuItem caption="Giddy Up" />
      </contextMenu>
    </csMenuItem>
    <csMenuItem caption="Kitty Cascade" >
      <contextMenu >
        <csMenuItem caption="Ignore me" />
        <csMenuItem caption="Do as you please" />
      </contextMenu>
    </csMenuItem>
  </contextMenu>
</group>
```

The following figure displays the popup menu that the previous example defines.

Figure 7–3: Popup Menu with Submenus



7.1.4 Cascading Menus

You can cascade menus by nesting <contextMenu> elements within <csMenuItem> elements. This cascading technique works for regular menus as well as for popup or context menus. See the examples in the opening section of this chapter, "Building Client Side Menus," and in the previous section, "Popup or Context Menus."

7.1.5 Button Bars

The <buttonBar> component is closely related to the client side menu components. A <buttonBar> may appear inside any sort of group container, and may contain one or more <buttonBarItem> components. The following <hgroup> element defines a button bar with three buttons, rollover tool tips, and selection actions (the alert code).

XML

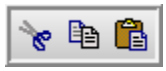
```
<hgroup id="bbArea" >
  <buttonBar id="editStuff" >
    <buttonBarItem icon="images/cut.png" caption="Cut"
                   onclick="alert('Cut requested');"  />
    <buttonBarItem icon="images/copy.png" caption="Copy"
                   onclick="alert('Copy requested');"  />
    <buttonBarItem icon="images/paste.png" caption="Paste"
                   onclick="alert('Paste requested');"  />
  </buttonBar>
</hgroup>
```

The style for this bar is defined in the XData Style section as follows:

```
#bbArea {
  background:#bbbbbb;
  background-image: url(images/grad-halfgray-10x30.png);
  background-repeat:repeat-x;
}
```

The following figure displays the button bar that this example defines.

Figure 7–4: Button Bar with Icons



For some image files that you can use as icons in button bars, see the section "Icons."

7.1.6 Icons

The following table lists the icons that are available in the csp/broker/images directory, which is a subdirectory of the Caché installation directory. The gray background color shown for icons in the table is just an example; the icons display over whatever background color the designer chooses for the containing component.

Icon images may be referenced using the *icon*, *altIcon*, or *disabledIcon* attributes in Zen components such as <buttonBarItem> or <csMenuItem>. When you reference these images, the prefix csp/broker is automatically implied. Reference them using this syntax:

images/*filename*

For example:

```
altIcon="images/cut.png"
```

Table 7–1: Zen Icon Library

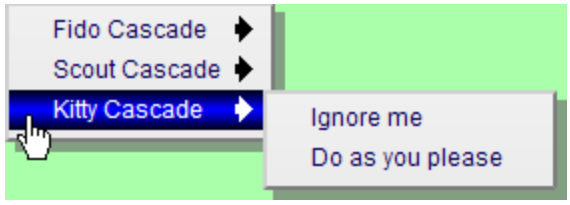
Icon	File Name	Meaning
	buildall.png	Build all or compile all files
	compile.png	Compile current file
	copy.png	Copy
	cut.png	Cut
	DownArrow.png	Down arrow, larger size, for light background colors
	DownArrowInv.png	Down arrow, larger size, for dark background colors
	SmDownArrow.png	Down arrow, smaller size, for light background colors
	SmDownArrowInv.png	Down arrow, smaller size, for dark background colors
	file.png	File or document
	find.png	Find or search
	goto.png	Go to
	inspect.png	Inspect
	SmLeftArrow.png	Left arrow, smaller size, for light background colors
	SmLeftArrowInv.png	Left arrow, smaller size, for dark background colors
	open.png	Open file, folder, or document
	output.png	Output
	paste.png	Paste
	print.png	Print
	redo.png	Redo previous operation
	redo_g.png	Redo previous operation, grayed out
	reload.png	Reload
	RtArrow.png	Right arrow, larger size, for light background colors
	RtArrowInv.png	Right arrow, larger size, for dark background colors
	SmRightArrow.png	Right arrow, smaller size, for light background colors
	SmRightArrowInv.png	Right arrow, smaller size, for dark background colors
	save.png	Save file, folder, or document

Icon	File Name	Meaning
	save_g.png	Save file, folder, or document, grayed out
	saveall.png	Save all files or documents
	saveas.png	Save current file or document to a new file
	undo.png	Undo previous operation
	undo_g.png	Undo previous operation, grayed out
	UpArrow.png	Up arrow, larger size, for light background colors
	UpArrowInv.png	Up arrow, larger size, for dark background colors
	SmUpArrow.png	Up arrow, smaller size, for light background colors
	SmUpArrowInv.png	Up arrow, smaller size, for dark background colors
	viewother.png	View other, such as the compiled routine for a class
	watch.png	Watch
	workspace.png	Workspace
	world.png	World, as in World Wide Web

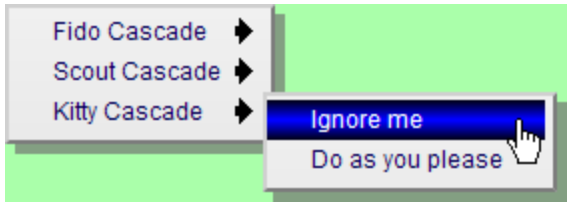
The *name*Inv.png images are used automatically in place of their *name*.png counterparts when Zen detects that it is rendering the icon against a dark background color. *name*.png icons are intended for use against light background colors. This behavior can be seen in the system default color scheme for Zen, as follows:

1. The dark arrow over the light background:

2. ...automatically changes to a light arrow when it is over a dark background:



3. ...and changes back to a dark arrow when it is over a light background:



The switch to using light or dark icons happens automatically, as long as you define the foreground and background colors of both <csMenuItem> and <csActiveMenuItem> using CSS. This allows Zen to calculate which icon yields optimum contrast with the current color. This is important even if the foreground or background color is being created using an image to create a gradient effect, as seen in the dark blue selected items in the previous examples.

7.2 <csMenuBar>

The <csMenuBar> component may appear inside any type of group container. The <csMenuBar> element defines a horizontal menu bar with one pick for each <csMenuBarItem> that is its direct child. <csMenuBar> may contain as many sibling <csMenuBarItem> components as are required to define the choices on the menu bar. For a <csMenuBar> example, see the section "Building Client Side Menus."

<csMenuBar> has the following attributes.

Attribute	Description
Zen group attributes	<csMenuBar> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.

7.3 <csMenuBarItem>

The <csMenuBarItem> component may appear only within a <csMenuBar> container. Each <csMenuBarItem> defines one choice on the menu bar. For an example, see the section "Building Client Side Menus."

<csMenuBarItem> has the following attributes.

Attribute	Description
Zen group attributes	<csMenuBarItem> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
caption	Text label to associate with this menu pick. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. This makes it easy to localize its 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 *invalidMessage* property from client-side or server-side code.
contextKey	If defined, the *contextKey* is a single character that identifies which letter in the *caption* text can be pressed to invoke the functionality of the menu pick. *contextKey* is not case-sensitive. The *contextKey* character is underlined when the *caption* text is displayed.

7.4 <contextMenu>

The <contextMenu> component may appear inside any type of group container, including <csMenuItem>, <vgroup>, <hgroup>, <form>, etc.

The group may contain other elements besides <contextMenu>, but to allow other elements to appear inside the group along with <contextMenu> you must set the containing group's *layout* attribute to "none". In the following example, a <button> appears in addition to the <contextMenu>, so the <form> has *layout* set to "none":

```
<form height="600" width="400" layout="none" >
  <button caption="MyButton"/>
  <contextMenu id="cm1">
    <csMenuItem caption="context"/>
    <csMenuItem caption="menu"/>
    <csMenuItem caption="1"/>
  </contextMenu>
</form>
```

To understand more about how <contextMenu> may be used to specify various types of menus, see the examples in the sections "Building Client Side Menus" and "Popup or Context Menus."

A <contextMenu> component may contain as many sibling <csMenuItem> and <csMenuSeparator> components as are required to define and organize the individual choices on the menu.

<contextMenu> has the following attributes.

Attribute	Description
Zen group attributes	<contextMenu> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.

7.5 <csMenuItem>

The <csMenuItem> component may appear only within a <contextMenu> container. <contextMenu> may contain as many sibling <csMenuItem> and <csMenuSeparator> components as are required to define and organize the choices on the menu.

<csMenuItem> has the following attributes.

Attribute	Description
Zen group attributes	In addition to its capabilities as a menu item, <csMenuItem> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.
altCaption	Text to display for the menu item under alternate conditions, such as when the user toggles the menu item. The normal alternate to *altCaption* is *caption*.
altIcon	URI of the image to display beside the *caption* text under alternate conditions, such as when the user toggles the menu item. If it is a pathname on the local server, the URI is relative to the Caché installation directory, as in: `altIcon="images/cut.png"` The normal alternate to *altIcon* is *icon*. For some available icons, see the section "Icons."
altKey	If defined, the *altKey* is a keyboard shortcut that can invoke the functionality of the menu item under alternate conditions, such as when the user toggles the menu item. The normal alternate to *altKey* is *key*.
caption	Text to display for the menu item under normal conditions. Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. This makes it easy to localize its 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 *invalidMessage* property from client-side or server-side code.
contextKey	If defined, the *contextKey* is a single character that identifies which letter in the *caption* text can be pressed to invoke the functionality of the menu item when it is active. *contextKey* is not case-sensitive: The *contextKey* character is underlined when the *caption* text is displayed. For detailed examples, see the section "Drop Down Menus."
icon	URI of the image to display beside the *caption* text under normal conditions. If it is a pathname on the local server, the URI is relative to the Caché installation directory, as in: `icon="images/cut.png"` For some available icons, see the section "Icons."

Attribute	Description
iconDisabled	URI of the image to display beside the *caption* text when this menu item is disabled but still visible (“grayed out).”. If it is a pathname on the local server, the URI is relative to the Caché installation directory, as in: `iconDisabled="images/cut.png"` For some available icons, see the section “Icons.”
key	If defined, the *key* is a keyboard shortcut that can invoke the functionality of the menu item when it is active, bypassing the need for a mouse click: For detailed examples, see the section “Drop Down Menus.”
onclick	Client-side JavaScript expression that runs each time the user clicks the mouse on this menu item. Generally this expression invokes a client-side JavaScript method defined in the page class. When providing a value for this attribute, use double quotes to enclose the value and single quotes (if needed) within the JavaScript expression. For example: `<csMenuItem onclick="alert('HEY');"/>` The JavaScript expression may contain Zen #()# runtime expressions.
toggleMode	If defined, and true, a checkmark indicates the toggled state of the menu pick. The checkmark displays when *toggleMode* is true. The default for *toggleMode* is false. *toggleMode* has the underlying data type %ZEN.Datatype.boolean. It has the value "true" or "false" in XData Contents, 1 or 0 in server-side code, true or false in client-side code.

7.6 <csMenuSeparator>

The <csMenuSeparator> component may appear only within a <contextMenu> container. <contextMenu> may contain as many sibling <csMenuItem> and <csMenuSeparator> components as are required to define and organize the choices on the menu. <csMenuSeparator> simply provides a horizontal line to separate sections of menu choices.

The <csMenuSeparator> component has no attributes. For example:

XML

```
<csMenuSeparator />
```

For more detailed examples, see the section “Drop Down Menus.”

7.7 <buttonBar>

The <buttonBar> component may appear inside any type of group container. <buttonBar> may contain as many sibling <buttonBarItem> components as are required to define the choices on the button bar. For an example, see the section “Button Bars.”

<buttonBar> has the following attributes.

Attribute	Description
Zen group attributes	<buttonBar> has the same style and layout attributes as other Zen groups. For descriptions, see "Group Layout and Style Attributes" in the "Zen Layout" chapter of *Using Zen*.

7.8 <buttonBarItem>

The <buttonBarItem> component may appear only within a <buttonBar> container. Each <buttonBarItem> defines one choice on the button bar. For an example, see the section "Button Bars."

<buttonBarItem> has the following attributes.

Attribute	Description
Zen component attributes	<buttonBarItem> 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	If defined, this is the rollover tooltip to display when button is enabled Although you can enter ordinary text for this attribute, it has the underlying data type %ZEN.Datatype.caption. This makes it easy to localize its 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 *invalidMessage* property from client-side or server-side code.
icon	URI of the image to display when the button is enabled. If it is a pathname on the local server, the URI is relative to the Caché installation directory, as in: `icon="images/cut.png"` For some available icons, see the section "Icons."
iconDisabled	URI of the image to display when the button is disabled but still visible ("grayed out)."). If it is a pathname on the local server, the URI is relative to the Caché installation directory, as in: `iconDisabled="images/cut.png"` For some available icons, see the section "Icons."

Attribute	Description
onclick	Client-side JavaScript expression that runs each time the user clicks the mouse on this button. Generally this expression invokes a client-side JavaScript method defined in the page class. When providing a value for this attribute, use double quotes to enclose the value and single quotes (if needed) within the JavaScript expression. For example: `<csMenuItem onclick="alert('HEY');"/>` The JavaScript expression may contain Zen #()# runtime expressions.

8

Client Side Library

The chapter "Client Side Layout Managers" describes the default client side layout functionality built into Zen. Zen also offers a client side library module that allows you to override Zen defaults to provide extremely fine control over all client side activities for your application.

When your Zen page uses components that this book explicitly identifies as client side components, all entities in the client side library module are available to you automatically. For the list of these components, see the previous two chapters, "Client Side Layout Managers" and "Client Side Menu Components." When this is the case there is no need to add any statements to your Zen page classes in order to call functions from the client side library. In other cases you can reference the library by adding it to the list of JavaScript files in the value of the JSINCLUDES parameter for the page class or application class. For example:

Class Member

```
Parameter JSINCLUDES = "zenCSLM.js";
```

This chapter provides a reference guide to the JavaScript methods and properties in the Zen client side library module. This chapter documents the support that the library provides for:

- Writing Custom Drag and Drop Methods
- Debugging Client Side Code

By convention, the names of entities defined in the client side library use the prefix ZLM followed by a dot, as in `ZLM.setDragAvatar` or `ZLM.getDragInnerDestination`. You can think of the ZLM prefix as a namespace that keeps the library entities distinct from other functions that you might define when working with Zen.

Note: Because the library is written in JavaScript, which does not facilitate generated class documentation, there is no Class Reference documentation for it in the InterSystems online documentation system. Refer to this chapter for all details.

8.1 Writing Custom Drag and Drop Methods

This section describes parts of the library that are important when you are customizing drag and drop methods for custom components, as outlined in the "Data Drag and Drop Methods" section in the chapter "Custom Components."

The following table describes the methods and properties of the JavaScript object called zenDragData that is automatically made available to subclasses of %ZEN.Component.component as a special variable called `dataDrag`.

Member	Meaning
`dataDrag.toString()`	This function returns a string that is equal to the value of the `dataDrag.text` property, if defined, otherwise the `dataDrag.value` property.
`dataDrag.value`	The logical value of the "source" control, which is the control whose value is being dragged.
`dataDrag.text`	The display value of the source control. If this is not available, the logical value is used.
`dataDrag.sourceComponent`	A pointer to the source control.
`dataDrag.targetComponent`	The logical value of the "target" control, which is the control onto which the value is being dropped.
`dataDrag.sourceItem`	A pointer to a specific data field within the source control. For example, if the control is a list box it may contain several list entries. Only one of these entries provides the data item that is being dragged.
`dataDrag.targetItem`	A pointer to a specific data field within the target control. This is the field onto which the data item is being dropped. For example, if the control is a list box this identifies a specific item within the list.

The Zen client side library module provides the following functions to support custom drag and drop methods.

8.1.1 ZLM.getDragData

```
ZLM.getDragData()
```

Returns a pointer to the `dragData` JavaScript object whose structure is described in the section "Data Drag and Drop Methods." This object contains all the information needed to execute a drag and drop operation.

8.1.2 ZLM.getDragDestination

```
ZLM.getDragDestination()
```

Returns a pointer to the enclosing for the Zen element where the current drag operation ended.

8.1.3 ZLM.getDragInnerSource

```
ZLM.getDragInnerSource()
```

Returns a pointer to the innermost enclosing element that initiated the current drag operation. This is a descendant of the enclosing for the Zen element that would be returned by `ZLM.getDragSource`.

8.1.4 ZLM.getDragInnerDestination

```
ZLM.getDragInnerDestination()
```

Returns a pointer to the innermost enclosing element where the current drag operation ended. This is a descendant of the enclosing for the Zen element that would be returned by `ZLM.getDragDestination`.

8.1.5 ZLM.getDragSource

`ZLM.getDragSource()`

Returns a pointer to the enclosing for the Zen element that initiated the current drag operation.

8.1.6 ZLM.setDragAvatar

`ZLM.setDragAvatar(`*newDiv*`)`

Allows a user-defined **dragStartHandler** to set the icon to use as the drag avatar of the object being dragged. The *newDiv* argument is the pointer to an enclosing object, which can be set up for the call to **ZLM.setDragAvatar** as shown in the following example, and in other examples in the section "Data Drag and Drop Methods":

Class Member

```
Method dragStartHandler(dragData) [ Language = javascript ]
{
  // get drag data
  if (!this.getDragData(dragData)) {
    return false;
  }

  // avatar
  var icon = this.getEnclosingDiv().cloneNode(true);
  icon.style.position="absolute";
  icon.style.border ="1px solid darkgray";
  icon.style.background ="#D0D0F0";
  ZLM.setDragAvatar(icon);

  return true;
}
```

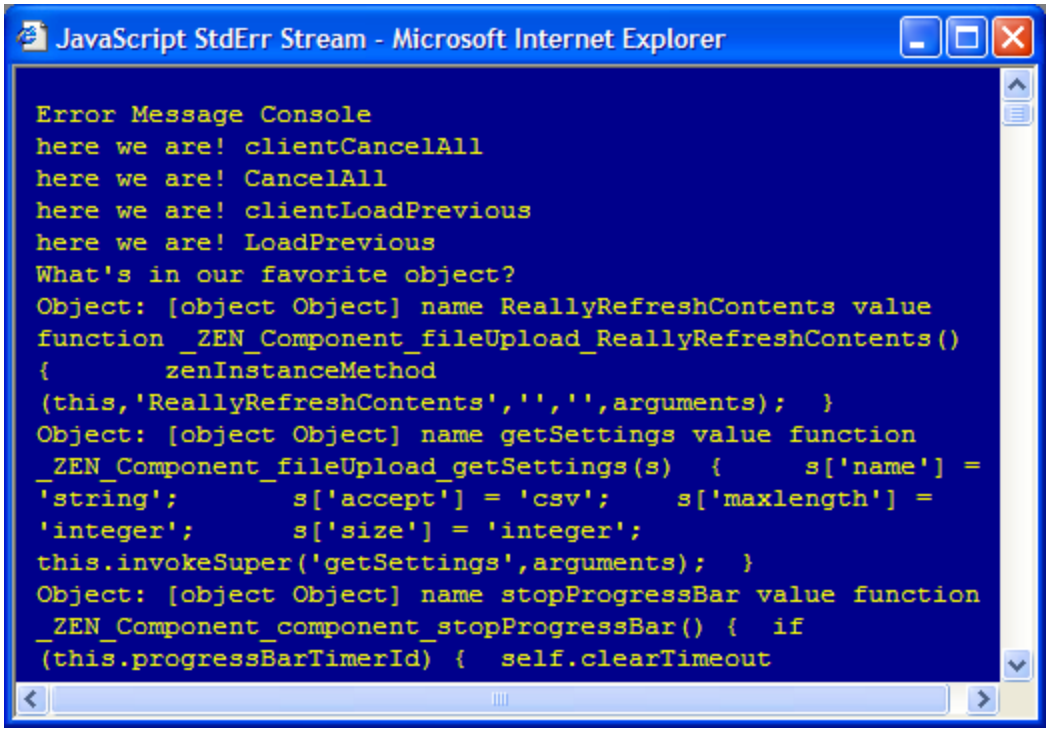
8.1.7 ZLM.setDragCaption

`ZLM.setDragCaption(`*caption*`)`

Allows a user-defined **dragStartHandler** to set the caption string of the object being dragged. The *caption* argument is a text string.

8.2 Debugging Client Side Code

This topic describes a set of Zen client side library module functions that support debugging by logging messages to a console window devoted to Zen client side activities. The following is a sample message console window. The output in this example reflects calls to **ZLM.cerr** and **ZLM.dumpObj**. These are only two of the available functions.



If a message console window is not already open as a result of a previous call to **ZLM.showMsgConsole**, any of the logging functions described in this section has the effect of opening the message console window. If there is a message console window already open, the logging function keeps writing to that window.

The logging functions are extremely useful in that they can be called from client-side JavaScript methods and from server-side Zen methods using &js<> syntax; that is:

```
&js<ZLM.cerr("We are now in a ZenMethod");>
```

This way, activities on both the server and the client side can write to the same message console window, showing the actual sequence of activities on both sides. The window stays open until full control returns to the server, for example when a form is submitted and its server-only method **%OnSubmit** executes. At that time the window automatically closes.

The Zen client side library module provides the following logging functions.

8.2.1 ZLM.showMsgConsole

`ZLM.showMsgConsole()`

Opens an extra window to serve as an error console. Initializes this window to respect whitespace and to use a monospaced font. As a debugging tool, this window is a useful alternative to a series of JavaScript **alert** calls which require you to repeatedly dismiss popup windows when messages display. The window stays open and displays all text provided by subsequent calls to **ZLM.cerr**.

8.2.2 ZLM.cerr

`ZLM.cerr(`*msg*`)`

Output a string to the message console window for debugging purposes. The *msg* argument is a text string. The following line can appear in a client-side JavaScript method:

```
ZLM.cerr("We are now in a client side method");
```

8.2.3 ZLM.dumpObj

`ZLM.dumpObj(`*obj*`)`

Write all known properties of a given object, as well as their current values, to the message console window.

The input argument *obj* can be any valid JavaScript object, including a Zen component or a JavaScript object such as an array, function, or custom data structure. The advantage of using **ZLM.dumpObj** to view the contents of an array is that you do not need to set up any iteration over the array. Simply input the array object to **ZLM.dumpObj**.

obj can identify a component on the Zen page, or the page itself. You can retrieve a valid *obj* value for a component with the client-side function **zen** and the *id* specified in the component definition. For example, if you have:

XML

```
<button id="myFavoriteObject" />
```

Then you can dump its current properties as follows:

```
ZLM.cerr("What's in my favorite object?");
ZLM.dumpObj(zen("myFavoriteObject"));
```

Alternatively, *obj* can be a pointer to the enclosing for the Zen element whose contents are of interest. You can retrieve this value by first invoking the client-side function **zen** with the *id* specified in the component definition, and then invoking the client-side component method **getEnclosingDiv**. The result is a value that you can input to the **ZLM.dumpObj** function. For example:

```
var comp = zen("thatComponent");
var div = comp.getEnclosingDiv();
ZLM.dumpObj(div);
```

8.2.4 ZLM.dumpDOMTreeGeometry

`ZLM.dumpDOMTreeGeometry(`*root*`)`

Write the nesting structure and base geometry of a tree of DOM nodes to the message console window. *root* is the root of the tree that you wish to display. *root* can be any valid DOM node from a JavaScript perspective. This would be any part of the HTML source that is associated with an HTML tag such as , , <input>, <p>, etc. — basically the web building blocks for drawing the HTML output using the component's %DrawHTML method.

To display the entire DOM tree, use `document.body` as the *root*; for example:

```
ZLM.cerr("What does the DOM for the entire page look like right now?");
ZLM.dumpDOMTreeGeometry(document.body);
```

Alternatively, *root* can be a pointer to the enclosing for the Zen component whose DOM structure is of interest. You can retrieve this value by first invoking the client-side function **zen** with the *id* specified in the component definition, and then invoking the client-side component method **getEnclosingDiv**. The result is a value that you can input to the **ZLM.dumpDOMTreeGeometry** function. For example:

```
var comp = zen("secondGroupFromLeft");
var div = comp.getEnclosingDiv();
ZLM.dumpDOMTreeGeometry(div);
```

The call to **getEnclosingDiv** returns an HTML element that we use as a wrapper around Zen components that might have multiple DOM nodes associated with them. This convention is designed to make sure that the DOM tree branches off in an orderly fashion. If the return value is used as input to **ZLM.dumpDOMTreeGeometry** you get a snapshot of the HTML structure created by the call to **%DrawHTML** for this component, plus any modifications to that structure resulting from applied stylesheets and/or "onload" client-side post-processing.

8.2.5 ZLM.dumpElementStyle

`ZLM.dumpElementStyle(`*div*`)`

Report all known style properties of a given component to the message console window. *div* is a pointer to the enclosing for the Zen component whose style properties are of interest.

You can retrieve the value of *div* by first invoking the client-side function **zen** with the *id* specified in the component definition, and then invoking the client-side component method **getEnclosingDiv**. The result is a value that you can input to the **ZLM.dumpElementStyle** function. For example:

```
var comp = zen("advancedOptions");
var div = comp.getEnclosingDiv();
ZLM.dumpElementStyle(div);
```