1. The ZK User Interface Markup Language

The ZK User Interface Markup Language (ZUML) is based on XML. Each XML element describes what component to create.

XML

This section provides the most basic concepts of XML to work with ZK. If you are familiar with XML, you could skip this section. If you want to learn more, there are a lot of resources on the Internet, such as http://www.w3schools.com/xml/xml_whatis.asp and http://www.xml.com/pub/a/98/10/guide0.html.

XML is a markup language much like HTML but with stricter and cleaner syntax. It has several characteristics worth to notice.

Elements Must Be Well-formed

First, each element must be closed. They are two ways to close an element as depicted below. They are equivalent.

<table>
<thead>
<tr>
<th>Close by an end tag:</th>
<th>&lt;window&gt;&lt;/window&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close without an end tag:</td>
<td>&lt;window/&gt;</td>
</tr>
</tbody>
</table>

Second, elements must be properly nested.

| Correct:          | <window>         |
|                  | <groupbox>      |
|                  |   Hello World!  |
|                  | </groupbox>     |
|                  | </window>       |
| Wrong:           | <window>        |
|                  | <groupbox>      |
|                  |   Hello World!  |
|                  | </window>       |
|                  | </groupbox>     |

Special Character Must Be Replaced

XML uses &lt; to denote an element, so you have to replace special characters. For example, you have to use \&lt; to represent the &lt; character.
<table>
<thead>
<tr>
<th>Special Character</th>
<th>Replaced With</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
</tbody>
</table>

It is interesting to notice that backslash (\) is not a special character, so you don't need to escape it at all.

**Attribute Values Must Be Specified and Quoted**

| Correct:          | width="100%"  
|                  | checked="true" |
| Wrong:            | width=100%     
|                  | checked        |

**Comments**

A comment is used to leave a note or to temporarily edit out a portion of XML code. To add a comment to XML, use <!-- and --> to escape them. Note that you may not include “--” anywhere within an XML comment.

```
<window>
<!-- this is a comment and ignored by ZK -->
</window>
```

**Conditional Evaluation**

The evaluation of an element could be conditional. By specifying the if or unless attribute or both, developers can control whether to evaluate the associated element.

In the following example, the window component is created only if a is 1 and b is not 2. If an element is ignored, all of its child elements are ignored, too.

```
<window if="${a==1}" unless="${b==2}">
  ...
</window>
```
Iterative Evaluation

The evaluation of an element could be iterative. By specifying a collection of objects to the `forEach` Attribute, developers could control how many time of the associated element shall be evaluated. For sake of description, we call an element is an iterative element if it is assigned with the `forEach` attribute.

In the following example, the list item is created three times. Notice that you have to use EL expression to specify the collection.

```xml
<variable name="grades">
    <attribute name="value">
        new String[] {"Best", "Better", "Good"}
    </attribute>
</variable>

$listbox$
  <listitem forEach="${grades}" label="${each}"/>
$\end{listbox}$

The iteration depends on the type of the specified value of the `forEach` attribute.

- If `java.util.Collection`, it iterates each element of the collection.
- If `java.util.Map`, it iterates each `Map.Entry` of the map.
- If `java.util.Iterator`, it iterates each element from the iterator.
- If `java.util Enumeration`, it iterates each element from the enumeration.
- If `Object[]`, `int[]`, `short[]`, `byte[]`, `char[]`, `float[]` or `double[]` is specified, it iterates each element from the array.
- If null, nothing is generated (it is ignored).
- If none of above types is specified, the associated element will be evaluated once as if a collection with a single item is specified.

```xml
$listbox$
  <listitem forEach="grades" label="${each}"/>
$\end{listbox}$

The `each` Variable

During the evaluation, a variable called `each` is created and assigned with the item from the specified collection. In the above example, `each` is assigned with "Best" in the first iteration, then "Better" and finally "Good".

ZK will preserve the value of the `each` variable if it is defined before, and restore it after the
The **forEachStatus Variable**

The `forEachStatus` variable is an instance of `org.zkoss.ui.util.ForEachStatus`. It holds the information about the current iteration. It is mainly used to get the item of the enclosing element that is also assigned with the `forEach` attribute.

In the following example, we use nested iterative elements to generate two listboxes.

```xml
<variable name="classes"><attribute name="value">
    new String[] {"College", "Graduate"}
</attribute></variable>
<variable name="grades"><attribute name="value">
    new Object[] {
        new String[] {"Best", "Better"},
        new String[] {"A++", "A+", "A"}
    }
</attribute></variable>
<hbox>
    <listbox width="200px" forEach="${classes}">
        <listhead>
            <listheader label="${each}"/>
        </listhead>
        <listitem label="${forEachStatus.previous.each}: ${each}" 
            forEach="${grades[forEachStatus.index]}"/>
    </listbox>
</hbox>
```

<table>
<thead>
<tr>
<th>College</th>
<th>Graduate</th>
</tr>
</thead>
<tbody>
<tr>
<td>College: Best</td>
<td>Better: A++</td>
</tr>
<tr>
<td>College: Better</td>
<td>Better: A+</td>
</tr>
<tr>
<td></td>
<td>Better: A</td>
</tr>
</tbody>
</table>

**Load on Demand**

By default, ZK creates a component based on what are defined in a ZUML page one-by-one, when loading the page. However, we can defer the creation of a portion of components, until they become visible. This feature is called load-on-demand. It improves the performance, if there are many invisible components at the beginning.
Load-on-Demand with the **fulfill** Attribute

The simplest way to defer the creation of the child components is to use the **fulfill** attribute. For example, the **comboitem** component in the following code snippet will *not* be created, until the **combobox** component receives the **onOpen** event, indicating **comboitem** is becoming visible.

```xml
<combobox fulfill="onOpen">
  <comboitem label="First Option"/>
</combobox>
```

In other words, if a ZUML element is specified with the **fulfill** attribute, its child elements won't be processed until the event specified as the value of the **fulfill** attribute is received.

If the event to trigger the creation of children is targeted to another component, you can specify the target component's identifier after colon as depicted below.

```xml
<button id="btn" label="show"><attribute name="onClick">
  content.visible = true;
</attribute></button>
<div id="content" fulfill="btn.onClick">
  Any content created automatically when btn is clicked
</div>
```

If the components belong to different ID space, you can specify a path after the event name as follows.

```xml
<button id="btn" label="show"><attribute name="onClick">
  content.visible = true;
</attribute></button>
>window id="content" fulfill="../btn.onClick">
  Any content created automatically when btn is clicked
</window>
```
2. ZUML with the XUL Component Set

This chapter describes the set of XUL components. Unlike other implementation, XUL components of ZK is optimized for co-operating across Internet. Some components might not be totally compliant with XUL standards. For sake of convenience, we sometimes refer them as ZUL components.

Basic Components

Label

A label component represents a piece of text.

```
<window border="normal">
  Hello World
</window>
```

If you want to specify attribute to a label, you have to specify `<label>` explicitly as follows.

```
<window border="normal">
  <label style="color: red" value="Hello World"/>
</window>
```

**Tip:** ZUML is XML, not HTML, so it doesn't accept `&nbsp;`. However, you can use `&#160;` instead.

The `pre`, `hyphen`, `maxlength` and `multiline` Properties

You can control how a label is displayed with the `pre`, `hyphen`, and `maxlength` Properties. For example, if you specify `pre` to be true, all white spaces, such as new line, space and tab, are preserved.

<table>
<thead>
<tr>
<th>hyphen</th>
<th>pre</th>
<th>maxlenth</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>positive</td>
<td>Truncated the characters that exceeds the specified maxlenth.</td>
</tr>
<tr>
<td>true</td>
<td>any</td>
<td>positive</td>
<td>If the length of a word exceeds maxlenth, the word is hypernated.</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>any</td>
<td>maxlenth is ignored.</td>
</tr>
<tr>
<td>any</td>
<td>any</td>
<td>0</td>
<td>hyphen is ignored.</td>
</tr>
</tbody>
</table>
The multiline property is similar to the pre property, except it preserves only the new lines and the white spaces at the beginning of each line.

**Buttons**

There are two types of buttons: button and toolbarbutton. They behave similar except the appearance is different. The button component uses HTML BUTTON tag, while the toolbarbutton component uses HTML A tag.

You could assign a label and an image to a button by the label and image properties. If both are specified, the dir property controls which is displayed up front, and the orient property controls whether the layout is horizontal or vertical.

```xml
<button label="Left" image="/img/folder.gif" width="125px"/>
<button label="Right" image="/img/folder.gif" dir="reverse" width="125px"/>
<button label="Above" image="/img/folder.gif" orient="vertical" width="125px"/>
<button label="Below" image="/img/folder.gif" orient="vertical" dir="reverse" width="125px"/>
```

In addition to identifying images by URL, you could assign a dynamically generated image to a button by use of the setImageContent method. Refer to the following section for details.

**Tip:** The setImageContent method is supplied by all components that has the image property. Simplicity put, setImageContent is used for dynamically generated images, while image is used for images identifiable by URL.

**The onClick Event and href Property**

There are two ways to add behaviors to button and toolbarbutton. First, you could specify a listener for the onClick event. Second, you could specify an URL for the href
property. If both are specified, the `href` property has the higher priority, i.e., the onClick event won’t be sent.

```html
<button><attribute name="onClick"> do_something_in_Java(); </attribute></button>
<button href="/another_page.zul"/>
```

**The `sendRedirect` Method of the `org.zkoss.zk.ui.Execution` Interface**

When processing an event, you could decide to stop processing the current desktop and redirect to another page by use of the `sendRedirect` method. In other words, the following two buttons are equivalent (from user’s viewpoint).

```html
<button><attribute name="onClick">
    Executions.sendRedirect("another.zhtml");
</attribute></button>
<button href="another.zul"/>
```

Since the onClick event is sent to the server for processing, you could add more logic before invoking `sendRedirect`, such as redirecting to another page only if certain condition is satisfied.

On the other hand, the `href` property is processed completely at the client side. Your application won’t be noticed, when users clicks on the button.

**Radio and Radio Group**

A radio button is a component that can be turned on and off. Radio buttons are grouped together in a group, called `radiogroup`. Only one radio button with the same group may be selected at a time.

```html
<radiogroup onCheck="alert(self.selectedItem.label)">
    <radio label="Apple"/>
    <radio label="Orange"/>
    <radio label="Banana"/>
</radiogroup>
```

**Versatile Layouts**

You can mix `radiogroup` and `radio` to have the layout you want, as illustrated below.

```html
<radiogroup>
    <grid>
    <rows>
        <row><radio label="Apple" selected="true"/> Fruit, music or computer</row>
    </rows>
</grid>
</radiogroup>
```
The radio button belongs to the nearest ancestor radio group. You can even nest one radio group to another as follow. Each of them operate independently, though there might be some sort of visual overlap.

```xml
<grid>
  <rows>
    <row><radio label="Apple" selected="true"/> Fruit, music or computer</row>
    <row><radio label="Orange"/></row>
    <radio group>
      <radio label="Small"/>
      <radio label="Large" selected="true"/>
    </radio group>
    <row><radio label="Banana"/> <datebox/></row>
  </rows>
</grid>
```

**Image**

An image component is used to display an image at the browser. There are two ways to assign an image to an image component. First, you could use the `src` property to specify a URI where the image is located. This approach is similar to what HTML supports. It is useful if you want to display a static image, or any image that can be identified by URL.

```xml
<image src="/some/my.jpg"/>
```

**Locale Dependent Image**

Like using any other properties that accept an URI, you could specify "*" for identifying a Locale dependent image. For example, if you have different image for different Locales, you could use as follows.

```xml
<image src="/my*.png"/>
```

Then, assume one of your users is visiting your page with `de_DE` as the preferred Locale. Zk will try to locate the image file called `/my_de_DE.png`. If not found, it will try
and finally /my.png.

Second, you could use the `setContent` method to assign the content of an image into an `image` component directly. Once assigned, the image displayed at the browser is updated automatically. This approach is useful if an image is generated dynamically.

For example, you could generate a map for the location specified by a user as below.

Map: `<image id="image"/>`
Location: `<textbox><attribute name="onChange">
  if (self.value.length() > 0)
  {
    image.setContent(new MapImage(self.value));
  }
</attribute></textbox>`

In the above example, we assume you have a class called `MapImage` for generating a map of the specified location, which is so-called business logic.

Notice that the image component accepts the content only in the `org.zkoss.image.Image` interface. If the image generated by your tool is not in this format, you could use the `org.zkoss.image.AImage` class to wrap a binary array of data, a file or an input stream into the `Image` interface.

In traditional Web applications, caching a dynamically generated image is complicated. With the `image` component, you don’t need to worry about it. Once the content of an image is assigned, it belongs to the `image` component, and the memory it occupies will be released automatically after the `image` component is no long used.

**Tip:** If you want to display the contents, say PDF, other than image and audio, you could use the `iframe` component. Refer to the relevant section for details.

**Imagemap**

A `imagemap` component is a special image. It accepts whatever properties an `image` component accepts. However, unlike `image`, if a user clicks on the image, an `onClick` event is sent back to the server with the coordinates of the mouse position. In contrast, the `onClick` event sent by `image` doesn't contain the coordinates.

The coordinates of the mouse position are screen pixels counted from the upper-left corner of the image beginning with (0, 0). It is stored as instance of `org.zkoss.zk.ui.event.MouseEvent`. Once the application receives the `onClick` event, it could examine the coordinates of the mouse position from the `getX` and `getY` methods.

For example, if a user clicks 208 pixels over and 205 pixels down from the upper-left corner
of the image displayed from the following statement.

```html
<imagemap src="/img/sun.jpg"><attribute name="onClick">
    alert(event.x + ", " + event.y)
</attribute></imagemap>
```

Then, the user gets the result as depicted below.

![Image Map Example]

The application usually uses the coordinates to determine where a user has clicked, and then response accordingly.

**Area**

Instead of processing the coordinates by the application itself, developers could add the *area* components as the children of a *imagemap* component.

```html
<imagemap src="/img/sun.jpg">
    <attribute name="onClick">
        alert(event.area);
    </attribute>
    <area id="First" coords="0, 0, 100, 100"/>
    <area id="Second" shape="circle" coords="200, 200, 100"/>
</imagemap>
```

Then, the imagemap component will translate the coordinates of the mouse position to a logical name: the identifier of the area that users has clicked.

For example, if users clicks at (150, 150), then the user gets the result as depicted blow.
The **shape Property**

An area component supports three kinds of shapes: circle, polygon and rectangle. The coordinates of the mouse position are screen pixels counted from the upper-left corner of the image beginning with (0, 0).

<table>
<thead>
<tr>
<th>Shape</th>
<th>Coordinates / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>coords=&quot;x, y, r&quot;</td>
</tr>
<tr>
<td></td>
<td>where x and y define the position of the center of the circle and r is its radius in pixels.</td>
</tr>
<tr>
<td>polygon</td>
<td>coords=&quot;x1, y1, x2, y2, x3, y3...&quot;</td>
</tr>
<tr>
<td></td>
<td>where each pair of x and y define a vertex of the polygon. At least thee pairs of coordinates are required to defined a triangle. The polygon is automatically closed, so it is not necessary to repeat the first coordinate at the end of the list to close the region.</td>
</tr>
<tr>
<td>rectangle</td>
<td>coords=&quot;x1, y1, x2, y2&quot;</td>
</tr>
<tr>
<td></td>
<td>where the first coordinate pair is one corner of the rectangle and the other pair is the corner diagonally opposite. A rectangle is just a shortened way of specifying a polygon with four vertices.</td>
</tr>
</tbody>
</table>

If the coordinates in one area component overlap with another, the first one takes precedence.
Audio

An audio component is used to play the audio at the browser. Like image, you could use the src property to specify an URL of an audio resource, or the setContent method to specify a dynamically generated audio.

Depending on the browser and the audio plugin, developers might be able to control the play of an audio by the play, stop and pause methods. Currently, Internet Explorer with Media Player is capable of such controls.

Input Controls

A set of input controls are supported in the XUL component set: textbox, intbox, decimalbox, doublebox, datebox, combobox, and bandbox. They are used to let users input different types of data.

<zk>
	<textbox/>
	<datebox/>
</zk>

Tip: combobox and bandbox are special input boxes. They shares the common properties described here. Their unique features will be discussed later in the Comboboxes and Bandboxes section.

The type Property

You could specify the type property with password for the textbox components, such that what user has entered won't be shown.

Username: <textbox/>
Password: <textbox type="password"/>

The format Property

You could control the format of an input control by the format filed. The default is null. For datebox, it means yyyy/MM/dd. For intbox and decimalbox, it means no formating at all.

<datebox format="MM/dd/yyyy"/>
<decimalbox format="#,##0.##"/>

Like any other properties, you could change the format dynamically, as depicted below.
<datebox id="db"/>

<button label="set MM-dd-yyyy">
  <attribute name="onClick">
    db.setFormat("MM-dd-yyyy");
  </attribute>
</button>

**Mouseless Entry**

- **Alt+DOWN** to pop up the calendar.
- **LEFT, RIGHT, UP and DOWN** to change the selected day from the calendar.
- **ENTER** to activate the selection by copying the selected day to the **datebox** control.
- **Alt+UP or ESC** to give up the selection and close the calendar.

**Constraints**

You could specify what value to accept for input controls by use of the constraint property. It could a combination of no positive, no negative, no zero, no empty, no future, no past, no today, and a regular expression. The first three constraints are applicable only to **intbox** and **decimalbox**. The constraints of no future, no past, and no today are applicable only to **datebox**. The constraint of no empty is applicable to any type of components. The constraint of regular expressions is applicable only to String-type input components, such as **textbox**, **combobox** and **bandbox**.

To specify two or more constraints, use comma to separate them as follows.

```xml
<intbox constraint="no negative,no zero"/>
```

To specify a regular expression, you could have to use / to enclose the regular expression as follows.

```xml
<textbox constraint="/.+\@[a-z]+"/>
```

**Notes:**

- The above statement is XML, so do **not** use \ to specify a backslash. On the other hand, it is necessary, if writing in Java.

```java
new Textbox().setConstraint("/.+\@[a-z]+/");
```

1. It is allowed to mix regular expression with other constraints by separating them with comma.

You prefer to display an application dependent message instead of default one, you could append the constraint with colon and the message you want to display when failed.

```xml
<textbox constraint="/.+\@[a-z]+: e-mail address only"/>
<datebox constraint="no empty, no future: now or never"/>
```
Notes:

- The error message, if specified, must be the last element and start with colon.

- To support multilingual, you could use the `l` function as depicted in the **Internationalization** chapter.

```xml
<textbox constraint="/.+@[a-z]+/: ${c:l('err.email.required')}"/>
```

**Custom Constraints**

If you want more sophisticated constraint, you could specify an object which implements the `org.zkoss.zul.Constraint` interface.

```xml
<window title="Custom Constraint">
    <variable name="ctt"><attribute name="value">
        <![CDATA[
            new Constraint() {
                public void validate(Component comp, Object value) throws WrongValueException {
                    if (value == null || ((Integer)value).intValue() < 100) {
                        throw new WrongValueException(comp, "At least 100 must be specified");
                    }
                }
            }
        ]]> </attribute>
    </variable>
    <intbox constraint="${ctt}"/>
</window>
```

You could implement your constraint into a Java class, say `EmailValidator`, then:

```xml
<?taglib uri="/WEB-INF/tld/web/core.dsp.tld" prefix="c"?>
<textbox constraint="${c:new('EmailValidator')}"/>
```

**org.zkoss.zk.ui.WrongValueException**

In the above example, we use `org.zkoss.zk.ui.WrongValueException` to denote an error. As depicted, you have to specify the first argument with the component that causes the error, and then the second argument with the error message.

You could throw this exception anytime, such as when an `onChange` event is received as follows.

```
<textbox>
    <attribute name="onChange">
        if (!self.value.equals("good")) {
            self.value = "try again";
        }
    </attribute>
</textbox>
```
throw new WrongValueException(self, "Not a good answer!");

The onChange Event

An input control notifies the application with the onChange event if its content is changed by the user.

Notice that, when the onChange's event listener is invoked, the value has been set. Thus, it is too late if you want to reject illegal value in the onChange's event listener, unless you restore the value properly. Rather, it is recommended to use a constraint as described in the Custom Constraints section.

The onChanging event

An input control also notifies the application with the onChanging event, when user is changing the content.

Notice that, when the onChanging's listener is invoked, the value is not set yet. In other worlds, the value property still remain in the old value. To retrieve what the user is entering, you have to access the value property of the event as follows.

```
<grid>
  <rows>
    <row>The onChanging textbox:
      <textbox onChanging="copy.value = event.value"/>
    </row>
    <row>Instant copy:
      <textbox id="copy" readonly="true"></row>
  </rows>
</grid>
```

It is too early if you want to reject illegal value in the onChanging's event listener, because user may not complete the change yet. Rather, it is recommended to use a constraint as described in the Custom Constraints section.

Calendar

A calendar displays a 'flat' calendar and allows user to select a day from it.
The value Property and the onChange Event

Like input controls, calendar supports the value property to let developers set and retrieve the selected day. In addition, developers could listen to the onChange event to process it immediately, if necessary.

The compact Property

A calendar supports two different layouts and you can control it by use of the compact property.

```
<calendar compact="true"/>
```

The default value depends on the current Locale.

Progressmeter

A progress meter is a bar that indicates how much of a task has been completed. The value property must be in the range between 0 and 100.

```
<progressmeter value="10"/>
```

Slider

A slider is used to let user specifying a value by scrolling.

```
<slider id="slider" onScroll="Audio.setVolume(slider.curpos)"/>
```

A slider accepts a range of value starting from 0 to 100. You could change the maximal allowed value by the maxpos property.

Timer

A timer is an invisible component used to send the onTimer event to the server at the
specified time or period. You could control a timer by the \texttt{start} and \texttt{stop} methods.

```xml
<window title="Timer demo" border="normal">
  <label id="now"/>
  <timer id="timer" delay="1000" repeats="true"
      onTimer="now.setValue(new Date().toString());"/>
  <separator bar="true"/>
  <button label="Stops timer" attribute="name="onClick">
      timer.stop();
  </attribute></button>
  <button label="Starts timer" attribute="name="onClick">
      timer.start();
  </attribute></button>
</window>
```

**Paging**

A paging component is used to separate long content into multiple pages. For example, assume that you have 100 items and prefer to show 20 items at a time, then you can use the paging components as follows.

```xml
<paging totalSize="100" pageSize="20"/>
```

Then, when a user clicks on the hyperlinks, the \texttt{onPaging} event is sent with an instance of \texttt{org.zkoss.zul.event.PagingEvent} to the paging component. To decide which portion of your 100 items are visible, you shall add a listener to the paging component.

**Paging with List Boxes and Grids**

The \texttt{listbox} and \texttt{grid} component support the paging intrinsically, so you don't need to specify a paging component explicitly as above, unless you want to have different visual layout or to control multiple \texttt{listbox} and \texttt{grid} with one paging component.

Refer to the \textbf{Grids} section for more details.

**Windows**

A window is, like HTML DIV tag, used to group components. Unlike other components, a window has the following characteristics.

- A window is an owner of an ID space. Any component contained in a window, including itself, could be found by use of the \texttt{getFellow} method, if it is assigned with an identifier.
• A window could be overlapped, popup, and embedded.
• A window could be a modal dialog.

**Titles and Captions**

A window might have a title, a caption and a border. The title is specified by the `title` property. The caption is specified by declaring a child component called `caption`. All children of the `caption` component will appear to the right side of the title.

```xml
<window title="Demo" border="normal" width="350px">
  <caption>
    <toolbarbutton label="More"/>
    <toolbarbutton label="Help"/>
  </caption>
  <toolbar>
    <toolbarbutton label="Save"/>
    <toolbarbutton label="Cancel"/>
  </toolbar>
  What is your favorite framework?
  <radiogroup>
    <radio label="ZK"/>
    <radio label="JSF"/>
  </radiogroup>
</window>
```

You could also specify a label and an image to a caption, and then the appearance is as follows.

```xml
<window id="win" title="Main" border="normal" width="200px">
  <caption image="/img/coffee.gif" label="Hi there!"/>
  <checkbox label="Hello, World!"/>
  <window>
</window>
```

**The closable Property**

By setting the `closable` property to true, a close button is shown for the window, such that user could close the window by clicking the button. Once user clicks on the close button, an `onClose` event is sent to the window. It is processed by the `onClose` method of `Window`. Then, `onClose`, by default, detaches the window itself.

You can override it to do whatever you want. Or, you registered a listener to change the default behavior. For example, you might choose to hide rather than close.
In this example, this window hides itself when the close button is clicked.

Notice that `event.stopPropagation()` must be called to prevent `Window.onClose()` being called.

**Tip:** If the window is a popup, the `onOpen` event will be sent to the window with `open=false`, when the popup is closed due to user’s clicking outside of the window, or pressing ESC.

It is a bit confusing but `onClose` is sent to ask the server to detach or to hide the window. By default, the window is detached. Of course, the application can override it and do whatever it wants as described above.

On the other hand, `onOpen` is a notification. It is sent to notify the application that the client has hidden the window. The application cannot prevent it from be hidden, or change the behavior to be detached.

### The `sizable` Property

If you allow users to resize the window, you can specify `true` to the `sizable` property as follows. Once allowed, users can resize the window by dragging the borders.

```xml
<window id="win" title="Sizable Window" border="normal" width="200px" sizable="true">
    This is a sizable window.
    <button label="Change Sizable"><attribute name="onClick">
        win.sizable = !win.sizable;
    </attribute></button>
</window>
```

### The `onSize` Event

Once a user resizes the window, the `onSize` event is sent with an instance of `org.zkoss.zul.event.SizeEvent`. Notice that the window is resized before the `onSize` event is sent. In other word, the event serves as a notification that you generally ignore. Of course, you can do whatever you want in the event listener.

**Note:** If the user drags the upper or left border, the `onMove` event is also sent since the position is changed, too.

### The Style Class (`sclass`)

ZK supports four different style classes for window: `embedded`, `overlapped`, `popup` and
wndcyan. Of course, you can add more if you want.

By default, the `sclass` property is the same as the window mode, so windows in different modes appear differently. To change the appearance, simply assign a value to the `sclass` property as illustrated in the following example.

```xml
<hbox>
  <window title="Embedded Style" border="normal" width="200px">
    Hello, Embedded!
  </window>
  <window title="Cyan Style" sclass="wndcyan" border="normal" width="200px">
    Hello, Cyan!
  </window>
  <window title="Popup Style" sclass="popup" border="normal" width="200px">
    Hello, Popup!
  </window>
  <window title="Modal Style" sclass="modal" border="normal" width="200px">
    Hello, Modal!
  </window>
</hbox>
```

The `contentStyle` Property

You can customize the look and feel of the content block of the window by specifying the `contentStyle` property.

```xml
<window title="My Window" border="normal" width="200px"
  contentStyle="background:yellow">
  Hello, World!
</window>
```

Scrollable Window

A typical use of `contentType` is to make a window scrollable as follows.
Borders

The border property specifies whether to display a border for window. The default style sheets support only normal and none. The default value is none.

Of course, you can provide additional style class. For example,

```xml
<zk>
  <style>
    div.wc-embedded-dash {
      padding: 2px; border: 3px dashed #aab;
    }
  </style>
  <window title="My Window" border="dash" width="200px">
    Hello, World!
  </window>
</zk>
```

where wc-embedded-dash defines the style of the inner box of the window. The style class is named by concatenating wc\(^1\), the sclass property and the border property together and separating them with dash (-). In this example, sclass is embedded since it is an embedded window and no explicit sclass is assigned (so the default sclass is used).

Overlapped, Popup, Modal, Highlighted and Embedded

A window could be in one of four different modes: overlapped, popup, modal, highlighted and embedded. By default, it is in the embedded mode. You could change the mode by use of the doOverlapped, doPopup, doModal, doHighlighted, and doEmbedded methods, depicted as follows.

```xml
<zk>
  <window id="win" title="Hi!" border="normal" width="200px">
    <caption>
```

\(^1\) wc for window content, while wt for window title.
Embedded

An embedded window is placed inline with other components. In this mode, you cannot change its position, since the position is decided by the browser.

Overlapped

An overlapped window is overlapped with other components, such that users could drag it around and developer could set its position by the `setLeft` and `setTop` methods.

In addition to `doOverlapped`, you can use the `mode` property as follows.

```
<window title="My Overlapped" width="300px" mode="overlapped">
</window>
```

Popup

A popup window is similar to overlapped windows, except it is automatically closed when
user clicks on any component other than the popup window itself or any of its descendants. As its name suggested, it is designed to implement popup windows.

Modal

A modal window (aka., a modal dialog) is similar to the overlapped windows, except it suspends the execution until one of the `endModal`, `doEmbedded`, `doOverlapped`, `doHighlighted`, and `doPopup` methods is called.

In addition to suspending the execution, it disables components not belonging to the modal window.

A modal window is positioned automatically at the center of the browser, so you cannot control its position.

Highlighted

A highlighted window is similar to the overlapped windows, except the visual effect is the same as the modal windows. In other words, a highlighted window is positioned at the center of the browsers, and components not belonging to the highlighted window are disabled.

However, it does not suspend the execution. Like the overlapped windows, the execution continues to the next statement once the mode is changed. For example, `f1()` is called only after `win1` is closed, while `g()` is called immediately after `win2` becomes highlighted.

```java
win1.doModal(); //the execution is suspended until win1 is closed
f1();

win2.doHighlighted(); //the execution won't be suspended
g1()
```

The highlighted window is aimed to substitute the modal window, if you prefer not to use or suspend the event processing thread. Refer to the Use the Servlet Thread to Process Events section in the Advanced Features chapter.

Modal Windows and Event Listeners

Unlike other modes, you can only put a window into the modal mode in an event listener. In other words, you can invoke `doModal()` or `setMode("modal")` in an event listener.
wnd.visible = false;
</attribute></button>
</window>
<button label="do it"><attribute name="onClick">
  wnd.doModal();
</attribute></button>
</zk>

On the other hand, the following is wrong if it executes in the **Component Creation Phase**

```zul
//t1.zul
<window title="My Modal" width="300px" closable="true" mode="modal">
</window>
```

It will cause the following result if you browse it directly.

**HTTP Status 500 -**

```
com.potix.zk.ui.KronValueException: doModal() and setMode("modal") can only be called in an event listener, not in page loading
com.potix.zk.ui.html.Window.doModal(Window.java:249)
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
java.lang.reflect.Method.invoke(Unknown Source)
bsh.reflect.invokeMethod(Unknown Source)
bsh.reflect.invokeObjectMethod(Unknown Source)
bsh.Name.invokeMethod(Unknown Source)
bsh.BBBPrimaryExpression.eval(Unknown Source)
bsh.BBBPrimaryExpression.eval(Unknown Source)
bsh.interpreter.eval(Unknown Source)
```

If you need to create a modal window in page loading, you can post the `onModal` event.

**Note:** the following codes execute correctly even if `t1.zul` sets the window’s mode to modal directly (as shown above). Why? It executes in an event listener (for `onClick`).

---

2 Refer to the **Component Lifecycle** chapter.

3 Assume Tomcat is used.
The **position** Property

In addition to the **left** and **top** properties, you can control the position of an overlapped/popup/modal window by use of the **position** property. For example, the following code snippet positions the window to the right-bottom corner.

```xml
<window width="300px" mode="overlapped" position="right,bottom">
...
</window>
```

The value of the **position** property can be a combination of the following constants by separating them with comma (,).

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>center</td>
<td>Position the window at the center. If <strong>left</strong> or <strong>right</strong> is also specified, it means the vertical center. If <strong>top</strong> or <strong>bottom</strong> is also specified, it means the horizontal center. If none of <strong>left</strong>, <strong>right</strong>, <strong>top</strong> and <strong>bottom</strong> is specified, it means the center in both directions. Both the <strong>left</strong> and <strong>top</strong> property are ignored.</td>
</tr>
<tr>
<td>left</td>
<td>Position the window at the left edge. The <strong>left</strong> property is ignored.</td>
</tr>
<tr>
<td>right</td>
<td>Position the window at the right edge. The <strong>left</strong> property is ignored.</td>
</tr>
<tr>
<td>top</td>
<td>Position the window at the top. The <strong>top</strong> property is ignored.</td>
</tr>
<tr>
<td>bottom</td>
<td>Position the window at the bottom. The <strong>top</strong> property is ignored.</td>
</tr>
</tbody>
</table>

By default, its value is null. That is, the overlapped and popup window is positioned by the **left** and **top** properties, while the modal window is positioned at the center.
Common Dialogs

The XUL component set supports the following common dialogs to simplify some common tasks.

The Message Box

The `org.zkoss.zul.Messagebox` class provides a set of utilities to show message boxes. It is typically used to alert user when an error occurs, or to prompt user for a decision.

```java
if (Messagebox.show("Remove this file?", "Remove?", Messagebox.YES | Messagebox.NO, Messagebox.QUESTION) == Messagebox.YES) {
    ...//remove the file
}
```

Since it is common to alert user for an error, a global function called `alert` is available. The `alert` function is a shortcut of the `show` method in the `Messagebox` class. In other words, the following two statements are equivalent.

```java
alert("Wrong");
Messagebox.show("Wrong");
```

Notice that `Messagebox` is a modal window so it shares the same constraint: executable only in an event listener.

The File Upload Dialog

The `org.zkoss.zul.Fileupload` class provides a set of utilities to prompt a user for uploading file(s) from the client to the server. Once one of the `get` methods is called, a file upload dialog is shown at the browser to prompt the user for specifying file(s) for uploading. It won’t return until user has uploaded a file or presses the cancel button.

```xml
<window title="Fileupload Demo" border="normal">
    <image id="image"/>
    <button label="Upload">
        <attribute name="onClick">
            Object media = Fileupload.get();
            if (media instanceof org.zkoss.image.Image)
                image.setContent(media);
            else if (media != null)
                Messagebox.show("Not an image: "+media, "Error",
                    Messagebox.OK, Messagebox.ERROR);
        </attribute>
    </button>
</window>
```
Upload Multiple Files at Once

If you allow users to upload multiple files at once, you can specify the maximal allowed number as follows.

```zul
<window title="fileupload demo" border="normal">
  <button label="Upload">
    <attribute name="onClick"><![CDATA[{
      Object media = Fileupload.get(5);
      if (media != null) {
        for (int j = 0; j < media.length; ++j) {
          if (media[j] instanceof org.zkoss.image.Image) {
            Image image = new Image();
            image.setContent(media[j]);
            image.setParent(pics);
          } else if (media[j] != null) {
            MessageBox.show("Not an image: " + media[j], "Error",
              MessageBox.OK, MessageBox.ERROR);
          }
        }
      }
    }]]></attribute>
  </button>
  <vbox id="pics"/>
</window>
```

The fileupload Component

The fileupload component is not a modal dialog. Rather, it is a component, so it is placed inline with other components.
Note: In addition to providing the static `get` methods for opening the file upload dialogs, `org.zkoss.zul.Fileupload` itself is a component. It is the so-called fileupload component.

For example,

```xml
<image id="img"/>
Upload your hot shot:
<fileupload onUpload="img.setContent(event.media)"/>
```

**The onUpload Event**

When the Upload button is pressed, the `onUpload` event is sent with an instance of the `org.zkoss.zk.ui.event.UploadEvent` event. You can then retrieve the content of the upload files by use of the `getMedia` or `getMedias` methods.

Notice that `getMedia` and `getMedias` return null to indicate that no file is specified but the Upload button is pressed.

**The onClose Event**

In addition to `onUpload`, the `onClose` event is sent to notify that either the Upload button or the Cancel button is pressed. By default, it simply invalidates the `fileupload` component, i.e., all fields are cleaned up and redrawn. If you listen to this event to have the custom behavior.

**The File Download Dialog**

The `org.zkoss.zulFiledownload` class provides a set of utilities to prompt a user for downloading a file from the server to the client. Unlike the `iframe` component that displays the file in the browser window, a file download dialog is shown at the browser if one of the `save` methods is called. Then, the user can specify the location in his local file system to save the file.

```xml
<button label="Download download.html">
  <attribute name="onClick">{ java.io.InputStream is = desktop.getWebApp().getResourceAsStream("/test/download.html");
  if (is != null)
    Filedownload.save(is, "text/html", "download.html");
  else
    alert("/test/download.html not found");
  }</attribute>
</button>
```
The Box Model

Components: vbox, hbox and box.

The box model of XUL is used to divide a portion of the display into a series of boxes. Components inside of a box will orient themselves horizontally or vertically. By combining a series of boxes and separators, you can control the layout of the visual representation.

A box can lay out its children in one of two orientations, either horizontally or vertically. A horizontal box lines up its components horizontally and a vertical box orients its components vertically. You can think of a box as one row or one column from an HTML table.

Some examples are shown as follows.

```xml
<zk>
  <vbox>
    <button label="Button 1"/>
    <button label="Button 2"/>
  </vbox>
  <hbox>
    <button label="Button 3"/>
    <button label="Button 4"/>
  </hbox>
</zk>
```

The hbox component is used to create a horizontally oriented box. Each component placed in the hbox will be placed horizontally in a row. The vbox component is used to create a vertically oriented box. Added components will be placed underneath each other in a column.

There is also a generic box component which defaults to horizontal orientation, meaning that it is equivalent to the hbox. However, you can use the orient property to control the orientation of the box. You can set this property to the value horizontal to create a horizontal box and vertical to create a vertical box.

Thus, the two lines below are equivalent:

```xml
<vbox>
  <box orient="vertical">
```

You can add as many components as you want inside a box, including other boxes. In the case of a horizontal box, each additional component will be placed to the right of the previous one. The components will not wrap at all so the more components you add, the wider the window will be. Similarly, each element added to a vertical box will be placed underneath the previous one.
The **spacing** Property

You could control the spacing among children of the **box** control. For example, the following example puts 5em at both the upper margin and the lower margin. Notice: the total space between two input fields is 10em.

```xml
<vbox spacing="5em">
  <textbox/>
  <datebox/>
</vbox>
```

Another example illustrated an interesting layout by use of zero spacing.

```xml
<window title="Box Layout Demo" border="normal">
  <hbox spacing="0">
    <window border="normal">0</window>
  </hbox>
  <vbox spacing="0">
    <hbox spacing="0">
      <window border="normal">1</window>
      <window border="normal">2</window>
    </hbox>
    <vbox spacing="0">
      <window border="normal">3</window>
      <window border="normal">4</window>
    </vbox>
  </hbox>
</window>
```

The **widths and heights** Properties

You can specify the width for each cell of **hbox** with the **widths** property as follows.
The value is a list of widths separated by comma. If any value is missed, no width is generated for the corresponding cell and the real width is up to the browser.

Similarly, you can specify the heights for each cell of vbox with the heights property. Actually, these two properties are the same since the orientation of a box can be horizontal or vertical depending on the orient property.

Splitters

Components: splitter.

There may be times when you want to have two sections of a window where the user can resize the sections. This feature is accomplished by using a component called a splitter. It creates a skinny bar between two sections which allows either side to be resized.

A splitter must be put inside a box. When a splitter is placed inside a horizontal box (hbox), it will allow resizing horizontally. When a splitter is placed inside a vertical box (vbox), it will allow resizing vertically. For example,

And, the codes are as follows.

```xml
<vbox height="200px">
    Column 1-1: The left-top box. To know whether a splitter is collapsed, you can listen to the onOpen event.
    <splitter collapse="after"/>
    Column 1-2: You can enforce to open or collapse programming by calling setOpen method.
</vbox>
```
The collapse Property

It specifies which side of the splitter is collapsed when its grippy (aka., button) is clicked. If this property is not specified, the splitter will not cause a collapse (and the grippy won’t appear).

Allowed values and their meaning are as follows.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>No collapsing occurs.</td>
</tr>
<tr>
<td>before</td>
<td>When the grippy is clicked, the element immediately before the splitter in the same parent is collapsed so that its width or height is 0.</td>
</tr>
<tr>
<td>after</td>
<td>When the grippy is clicked, the element immediately after the splitter in the same parent is collapsed so that its width or height is 0.</td>
</tr>
</tbody>
</table>

The open Property

To know whether a splitter is collapsed, you can check the value of the open property (i.e., the isOpen method). To open or collapse programmingly, you can set the value of the open property (i.e., the setOpen method).

The onOpen Event

When a splitter is collapsed or opened by a user, the onOpen event is sent to the application.

Tab Boxes

Components: tabbox, tabs, tab, tabpanels andtabpanel.

A tab box allows developers to separate a large number of components into several groups, and show one group each time, such that the user interface won’t be too complicate to read. There is only one group (aka., a panel) is visible at the same time. Once the tab of an invisible group is clicked, it becomes visible and the previous visible group becomes invisible.

The generic syntax of tab boxes is as follows.
• **tabbox**: The outer box that contains the tabs and tab panels.

• **tabs**: The container for the tabs, i.e., a collection of tab components.

• **tab**: A specific tab. Clicking on the tab brings the tab panel to the front. You could put a label and an image on it.

• **tabpanels**: The container for the tab panels, i.e., a collection oftabpanel components.

• **tabpanel**: The body of a single tab panel. You would place the content for a group of components within a tab panel. The firsttabpanel corresponds to the first tab, the secondtabpanel corresponds to the second tab and so on.

The currently selected tab component is given an additional selected property which is set to true. This is used to give the currently selected tab a different appearance so that it will look selected. Only one tab will have a true value for this property at a time.

There are two ways to change the selected tab by Java codes. They are equivalent as shown below.

```java
tab1.setSelected(true);
tabbox.setSelectedTab(tab1);
```

Of course, you can assign true to the selected property directly.

```html
<tab label="My Tab" selected="true"/>
```

If none of tabs are selected, the first one is selected automatically.

**Nested Tab Boxes**

A tab panel could contain anything including another tab boxes.
The Accordion Tab Boxes

Tab boxes supports two molds: default and accordion. The effect of the accordion mold is as follows.
The **orient** Property

Developers can control whether the tabs are located by use of the **orient** property. By default, it is **horizontal**. You can change it to **vertical**, and the effect is as follows.

```html
<tabbox width="400px" orient="vertical">
  <tabs>
    <tab label="A"/>
    <tab label="B"/>
    <tab label="C"/>
    <tab label="D"/>
    <tab label="E"/>
  </tabs>
  <tabpanel>This is panel A</tabpanel>
  <tabpanel>This is panel B</tabpanel>
  <tabpanel>This is panel C</tabpanel>
  <tabpanel>This is panel D</tabpanel>
  <tabpanel>This is panel E</tabpanel>
</tabpanels>
</tabbox>
```

The **closable** Property

By setting the **closable** property to true, a close button is shown for the tab, such that user could close the tab and the corresponding tab panel by clicking the button. Once user clicks on the close button, an **onClose** event is sent to the tab. It is processed by the **onClose** method of **Tab**. Then, **onClose**, by default, detaches the tab itself and the corresponding tab panel.

See also window's **closable** property.

Load-on-Demand for Tab Panels

Like many other components, you can load the content of the tab panel only when it becomes visible. The simplest way is to use the **fulfill** attribute to defer the creation of the children of a tab panel.

```html
<tabbox>
  <tabs>
    <tab label="Preload" selected="true"/>
    <tab id="tab2" label="OnDemand"/>
  </tabs>
</tabbox>
```
If you prefer to create the children manually or manipulate the panel dynamically, you could listen to the `onSelect` event, and then fulfill the content of the panel when it is selected, as depicted below.

```xml
<tabbox id="tabbox" width="400" mold="accordion">
  <tabs>
    <tab label="Preload"/>
    <tab label="OnDemand"><attribute name="onSelect"><![CDATA[
      if (self.linkedPanel != null && self.linkedPanel.getChildren().isEmpty())
        new Label("Second panel is loaded").setParent(self.linkedPanel);
    ]]>"/></attribute></tab>
  </tabs>
  <tabpanels>
    <tabpanel>This panel is pre-loaded.</tabpanel>
    <tabpanel>This panel is loaded only tab2 receives the onSelect event.</tabpanel>
  </tabpanels>
</tabbox>
```

**Grids**

**Components:** grid, columns, column, rows and row.

A grid contains components that are aligned in rows like tables. Inside a grid, you declare two things, the columns, that define the header and column attributes, and the rows, that provide the content.

To declare a set of rows, use the `rows` component, which should be a child element of `grid`. Inside that you should add `row` components, which are used for each row. Inside the row element, you should place the content that you want inside that row. Each child is a column of the specific
Similarly, the columns are declared with the `columns` component, which should be placed as a child element of the grid. Unlike `row` is used to hold the content of each row, `column` declares the common attributes of each column, such as the width and alignment, and optional headers, i.e., label and/or image.

```xml
<grid>
  <columns>
    <column label="Type"/>
    <column label="Content"/>
  </columns>
  <rows>
    <row>
      <label value="File:"/>
      <textbox width="99%"/>
    </row>
    <row>
      <label value="Type:"/>
      <hbox>
        <listbox rows="1" mold="select">
          <listitem label="Java Files,(*.java)"/>
          <listitem label="All Files,(*.*)"/>
        </listbox>
        <button label="Browse..."/>
      </hbox>
    </row>
  </rows>
</grid>
```

**Scrollable Grid**

A grid could be scrollable if you specify the `height` property and there is not enough space to show all data.

```xml
<grid width="500px" height="130px">
  <columns>
    <column label="Head 1"/>
    <column label="Head 2" align="center"/>
    <column label="Head 3" align="right"/>
  </columns>
  <rows>
    <row>
```

ZK: Developer's Guide
<listbox mold="select">
    <listitem label="Faster"/>
    <listitem label="Fast"/>
    <listitem label="Average"/>
</listbox>
<datebox/>
<textbox rows="2"/>
</row>
<row>
    <checkbox checked="true" label="Option 1"/>
    <checkbox label="Option 2"/>
    <radiogroup>
        <radio label="Apple"/>
        <radio label="Orange" checked="true"/>
        <radio label="Lemon"/>
    </radiogroup>
</row>
<row>
    <checkbox checked="true" label="Option 1"/>
    <checkbox label="Option 2"/>
    <radiogroup orient="vertical">
        <radio label="Apple"/>
        <radio label="Orange" checked="true"/>
        <radio label="Lemon"/>
    </radiogroup>
</row>
</rows>
</grid>

**Sizable Columns**

If you allow users to resize the widths of columns, you can specify `true` to the `sizable` property of `columns` as follows. Once allowed, users can resize the widths of columns by dragging the border between adjacent `column` components.

```xml
<window>
    <grid>
        <columns id="cs" sizable="true">
            <column label="AA"/>
            <column label="BB"/>
            <column label="CC"/>
        </columns>
        <rows>
            <row>
                <label value="AA01"/>
                <label value="BB01"/>
                <label value="CC01"/>
            </row>
        </rows>
    </grid>
</window>
```
The **onColSize Event**

Once a user resizes the widths, the onColSize event is sent with an instance of org.zkoss.zul.event.ColSizeEvent. Notice that the column's width is adjusted before the onColSize event is sent. In other word, the event serves as a notification that you can ignore. Of course, you can do whatever you want in the event listener.

**Grids with Paging**

There are two ways to handle long content in a grid: scrolling and paging. The scrolling is enabled by specifying the `height` property as discussed in the previous section. The paging is enabled by specifying `paging` to the `mold` property. Once paging is enable, the grid separates the content into several pages and displays one page at a time as depicted below.

```html
<grid width="300px" mold="paging" pageSize="4">
  <columns>
    <column label="Left"/>
    <column label="Right"/>
  </columns>
  <rows>
    <row>
      <label value="Item 1.1"/></label value="Item 1.2"/>
    </row>
    <row>
      <label value="Item 2.1"/></label value="Item 2.2"/>
    </row>
  </rows>
</grid>
```
Once the paging mold is set, the grid creates an instance of the `paging` component as the child of the grid. It then takes care of paging for the grid it belongs to.

The **pageSize** Property

Once setting the `paging` mold, you can specify how many rows are visible at a time (i.e., the page size) by use of the `pageSize` property. By default, it is 20.

The **paginal** Property

If you prefer to put the `paging` component at different location or you want to control two or more grid with the same `paging` component, you can assign the `paginal` property explicitly. Note: if it is not set explicitly, it is the same as the `paging` property.

<table>
<thead>
<tr>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 5.1</td>
<td>Item 5.2</td>
<td>Item E.1</td>
<td>Item E.2</td>
</tr>
<tr>
<td>Item 6.1</td>
<td>Item 6.2</td>
<td>Item F.1</td>
<td>Item F.2</td>
</tr>
<tr>
<td>Item 7.1</td>
<td>Item 7.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The **paging** Property

It is a readonly property representing the child **paging** component that is created automatically to handling paging. It is null if you assign an external paging by the **paginal** property. You rarely need to access this property. Rather, use the **paginal** property.

The **onPaging** Event and Method

Once a user clicks the page number of the **paging** component, an **onPaging** event is sent the grid. It is then processed by the **onPaging** method. By default, the method invalidates, i.e., redraws, the content of **rows**.

If you want to implement "create-on-demand" feature, you can add a event listener to the grid for the **onPaging** event.

```java
grid.addEventListener(org.zkoss.zul.event.ZulEvents.ON_PAGING, new MyListener());
```

Sorting

Grids support the sorting of rows directly. To enable the ascending order for a particular column, you assign a **java.util.Comparator** instance to the **sortAscending** property of the column. Similarly, you assign a comparator to the **sortDescending** property to enable the descending order.

As illustrated below, you first implement a comparator that compares any two rows of the grid, and then assign its instances to the **sortAscending** and **sortDescending** properties. Notice: the **compare** method is called with two **org.zkoss.zul.Row** instance.

```java
// In a Java file called MyRowComparator.java:
class MyRowComparator implements Comparator {
    public MyRowComparator(boolean ascending) {
        ...
    }
    public int compare(Object o1, Object o2) {
        Row r1 = (Row)o1, r2 = (Row)o2;
        ....
    }
}
```
The `sortDirection` Property

The `sortDirection` property controls whether to show an icon at the client to indicate the order of a particular column. If rows are sorted before adding to the grid, you shall set this property explicitly.

```xml
<column sortDirection="ascending"/>
```

Then, it is maintained automatically by grids as long as you assign the comparators to the corresponding column.

The `onSort` Event

When you assign at least one comparator to a column, an `onSort` event is sent to the server if user clicks on it. The `column` component implements a listener to automatically sort rows based on the assigned comparator.

If you prefer to handle it manually, you can add your own listener to the column for the `onSort` event. To prevent the default listener to invoke the `sort` method, you have to call the `stopPropagation` method against the event being received. Alternatively, you can override the `sort` method, see below.

The `sort` Method

The `sort` method is the underlying implementation of the default `onSort` event listener. It is also useful if you want to sort the rows by Java codes. For example, you might have to call this method after adding rows (assuming not in the proper order).

```java
Row row = new Row();
row.setParent(rows);
row.appendChild(...);
...
if (!"natural".column.getSortDirection())
```
column.sort("ascending").equals(column.getSortDirection());

The default sorting algorithm is quick-sort (by use of the `sort` method from the `org.zkoss.zk.ui.Components` class). You might override it with your own implementation.

Note: the `sort` method checks the sort direction (by calling `getSortDirection`). It sorts the rows only if the sort direction is different. To enforce the sorting, do as follows.

column.setSortDirection("natural");
sort(myorder);

The above codes are equivalent to the following.

sort(myorder, true);

Special Properties

The spans Property

It is a list of integers, separated by comma, to control whether to span a cell over several columns. The first number in the list denotes the number of columns the first cell shall span. The second number denotes that of the second cell and so on. If the number is omitted, 1 is assumed.

For example,

```xml
<grid>
  <columns>
    <column label="Left" align="left"/>
    <column label="Center" align="center"/>
    <column label="Right" align="right"/>
    <column label="Column 4"/>
    <column label="Column 5"/>
    <column label="Column 6"/>
  </columns>
  <rows>
    <row>
      <label value="Item A.1"/>
      <label value="Item A.2"/>
      <label value="Item A.3"/>
      <label value="Item A.4"/>
      <label value="Item A.5"/>
      <label value="Item A.6"/>
    </row>
    <row spans="1,2,2">
      <label value="Item B.1"/>
      <label value="Item B.2"/>
      <label value="Item B.4"/>
      <label value="Item B.6"/>
    </row>
    <row spans="3">
      </row>
  </rows>
</grid>
```
More Layout Components

Separators and Spaces

Components: separator and space.

A separator is used to insert a space between two components. There are several ways to customize the separator.

1. By use of the `orient` property, you could specify a vertical separator or a horizontal separator. By default, it is a horizontal separator, which inserts a line break. On the other hand, a vertical separator inserts a white space. In addition, `space` is a variant of `separator` whose default orientation is vertical.

2. By use of the `bar` property, you could control whether to show a horizontal or vertical line between component.

3. By use of the `spacing` property, you could control the size of spacing.

```xml
<window>
  line 1 by separator
  <separator/>
  line 2 by separator
  <separator/>
  line 3 by separator<space bar="true"/>another piece
</window>
```
Group boxes

Components: groupbox.

A group box is used to group components together. A border is typically drawn around the components to show that they are related.

The label across the top of the group box can be created by using the caption component. It works much like the HTML legend element.

Unlike windows, a group box is not an owner of the ID space. It cannot be overlapped or popup.

```html
<groupbox mold="default" width="250px">
    <caption label="Fruits"/>
    <radiogroup>
        <radio label="Apple"/>
        <radio label="Orange"/>
        <radio label="Banana"/>
    </radiogroup>
</groupbox>
```

In addition to the default mold, the group box also supports the 3d mold. If you do not specify a mold, the 3d mold is used. If the 3d mold is used, it works similar to a simple-tab tab box. First, you could control whether its content is visible by the open property. Similarly, you could create the content of a group box when the onOpen event is received.

```html
<groupbox mold="3d" open="true" width="250px">
    <caption label="fruits"/>
    <radiogroup>
        <radio label="Apple"/>
        <radio label="Orange"/>
        <radio label="Banana"/>
    </radiogroup>
</groupbox>
```

The contentStyle Property and Scrollable Groupbox

The contentStyle property is used to specify the CSS style for the content block of the groupbox. Thus, you can make a groupbox scrollable by specify overflow:auto (or...
overflow:scroll) as follows.

```html
<groupbox mold="3d" width="150px" contentStyle="height:50px;overflow:auto">
  <caption label="fruits"/>
  <radiogroup onCheck="fruit.value = self.selectedItem.label" orient="vertical">
    <radio label="Apple"/>
    <radio label="Orange"/>
    <radio label="Banana"/>
  </radiogroup>
</groupbox>
```

**Note:** The `contentStyle` property is ignored if the default mold is used.

The height specified in the `contentStyle` property means the height of the content block, excluding the caption. Thus, if the groupbox is dismissed (i.e., the content block is not visible), the height of the whole groupbox will be shrunk to contain only the caption. On the other hand, if you specify the height for the whole groupbox (by use of the `height` property), only the content block disappears and the whole height remains intact, when dismissing the groupbox.

**Toolbars**

Component: `toolbar` and `toolbarbutton`.

A toolbar is used to place a series of buttons, such as toolbar buttons. The toolbar buttons could be used without toolbars, so a toolbar could be used without tool buttons. However, tool buttons change their appearance if they are placed inside a toolbar.

The toolbar has two orientation: horizontal and vertical. It controls how the buttons are placed.

```html
<toolbar>
  <toolbarbutton label="button1"/>
  <toolbarbutton label="button2"/>
</toolbar>
```

**Menu bars**

Component: `menubar`, `menupopup`, `menu`, `menuitem` and `menuseparator`.

A menu bar contains a collection of menu items and sub menus. A sub menu contains a collection of menu items and other sub menus. They, therefore, constructs a tree of menu items that user could select to execute.
An example of menu bars is as follows.

```xml
<menubar>
    <menu label="File">
        <menupopup>
            <menuitem label="New"/>
            <menuitem label="Open"/>
            <menuseparator/>
            <menuitem label="Exit"/>
        </menupopup>
    </menu>

    <menu label="Help">
        <menupopup>
            <menuitem label="Index"/>
            <menu label="About">
                <menupopup>
                    <menuitem label="About ZK"/>
                    <menuitem label="About Potix"/>
                </menupopup>
            </menu>
        </menupopup>
    </menu>
</menubar>
```

- **menubar**: The topmost container for a collection of menu items (**menuitem**) and menus (**menu**).
- **menu**: The container of a popup menu. It also defines the label to be displayed at part of its parent. When user clicks on the label, the popup menu appears.
- **menupopup**: A container for a collection of menu items (**menuitem**) and menus (**menu**). It is a child of **menu** and appears when the label of **menu** is clicked.
- **menuitem**: An individual command on a menu. This could be placed in a menu bar, or a popup menu.
- **menuseparator**: A separator bar on a menu. This would be placed in a popup menu.

**Execute a Menu Command**

A menu command is associated with a menu item. There are two ways to associate a command to it: the **onClick** event and the **href** property. If a event listener is added for a menu item for the **onClick** event, the listener is invoked when the item is clicked.
On the other hand, you could specify the \texttt{href} property to hyperlink to the specified URL when a menu item is clicked.

If both of the event listener and \texttt{href} are specified, they will be executed. However, when the event listener get executed in the server, the browser might already change the current URL to the specified one. Thus, all responses generated by the event listener will be ignored.

\textbf{Use Menu Items as Check Boxes}

A menu item could be used as a check box. The \texttt{checked} property denotes whether this menu item is checked. If checked, a check icon is appeared in front of the menu item.

In addition to programming the \texttt{checked} property, you could specify the \texttt{autocheck} property to be \texttt{true}, such that the \texttt{checked} property is toggled automatically when user clicks the menu item.

\textbf{The autodrop Property}

By default, the popup menu is opened when user clicks on it. You might change this to automatically popup menu when the mouse moves over it. This is done by setting the \texttt{autodrop} property to \texttt{true}.

\textbf{The onOpen Event}

When a menupopup is going to appear (or hide), an \texttt{onOpen} event is sent to the menupopup for notification. For sophisticated applications, you can defer the creation of the content of the menupopup or manipulate the content dynamically, until the \texttt{onOpen} event is received. Refer to the \textbf{Load on Demand} section in the \textit{ZK User Interface Markup Language} chapter for details.
More Menu Features

Like box, you could control the orientation of a menu by use of the orient property. By default, the orientation is horizontal.

Like other components, you could change the menu dynamically, including properties and creating sub menus. Refer to menu.zul under the test directory in zkdemo.

Context Menus

Components: popup and menupopup.

You can assign the ID of a popup or menupopup component to the context property of any XUL component, such that the popup or menupopup component is opened when a user right-clicks on it.

As depicted below, a context menu is enabled by simply assigning the ID to the context property. Of course, you can assign the same ID to multiple components.

```
<label value="Right Click Me!" context="editPopup"/>
<separator bar="true"/>
<label value="Right Click Me!" onRightClick="alert(self.value)"/>

<menupopup id="editPopup">
    <menuitem label="Undo"/>
    <menuitem label="Redo"/>
    <menu label="Sort">
        <menupopup>
            <menuitem label="Sort by Name" autocheck="true"/>
            <menuitem label="Sort by Date" autocheck="true"/>
        </menupopup>
    </menu>
</menupopup>
```

Notice that menupopup is not visible until a user right-clicks on a component associated with its ID.

**Trick:** If you just want to disable browser's default context menu, you can specify non-existent ID to the context property.

The popup component is a more generic popup than menupopup. You can place any kind of components inside of popup. For example,

```
<label value="Right Click Me!" context="any"/>
```
Customizable Tooltip and Popup Menus

In addition to open a popup when user right-clicks a component, ZK can open a popup under other circumstances.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>When user right clicks a component with the <code>context</code> property, the popup or <code>menupopup</code> component with the specified id is shown.</td>
</tr>
<tr>
<td>tooltip</td>
<td>When user move the mouse pointer over a component with the <code>tooltip</code> property, the popup or <code>menupopup</code> component with the specified id is shown.</td>
</tr>
<tr>
<td>popup</td>
<td>When user clicks a component with the <code>popup</code> property, the popup or <code>menupopup</code> component with the specified id is shown.</td>
</tr>
</tbody>
</table>

For example,

```xml
<window title="Context Menu and Right Click" border="normal" width="360px">
  <label value="Move Mouse Over Me!" tooltip="editPopup"/>
  <separator bar="true"/>
  <label value="Tooltip for Another Popup" tooltip="any"/>
  <separator bar="true"/>
  <label value="Click Me!" popup="editPopup"/>
  <menupopup id="editPopup">
    <menuitem label="Undo"/>
    <menuitem label="Redo"/>
    <menu label="Sort">
      <menupopup>
        <menuitem label="Sort by Name" autocheck="true"/>
        <menuitem label="Sort by Date" autocheck="true"/>
      </menupopup>
      </menu>
    </menu>
  </menupopup>
</window>
```
Notice that you can specify any identifier in the `popup`, `tooltip` and `context` properties, as long as they are in the same page. In other words, it is not confined by the ID space.

**The `onOpen` Event**

When a context menu, a tooltip or a popup is going to appear (or hide), an `onOpen` event is sent to the context, tooltip or popup menu for notification. The event is an instance of the `org.zkoss.zk.ui.event.OpenEvent` class, and you can retrieve the component that causes the context menu, tooltip or popup to appear by calling the `getReference` method.

To improve the performance, you defer the creation of the content until it becomes visible – i.e., until the `onOpen` event is received.

The simplest way to defer the creation of the content is to use the `fulfill` attribute as shown below.

```xml
<popup id="any" width="300px" fulfill="onOpen">
  <button label="Hi"/> <!-- whatever content -->
</popup>
```

Then, the content (the `Hi` button) won’t be created when the page is loaded. Rather, the content is created when the `onOpen` event is received at the first time.

If you prefer to dynamically manipulate the content in Java, you can listen to the `onOpen` event as depicted below.

```xml
<popup id="any" width="300px">
  <attribute name="onOpen">
    if (event.isOpen()) {
      if (self.getChildren().isEmpty()) {
        new Button("Hi").setParent(self);
        ...
      }
      if (event.getReference() instanceof Textbox) {
        //you can do component-dependent manipulation here
        ...
      }
    }
  </attribute>
</popup>
```
List Boxes

Components: listbox, listitem, listcell, listhead and listheader.

A list box is used to display a number of items in a list. The user may select an item from the list.

The simplest format is as follows. It is a single-column and single-selection list box.

```
<listbox>
    <listitem label="Butter Pecan"/>
    <listitem label="Chocolate Chip"/>
    <listitem label="Raspberry Ripple"/>
</listbox>
```

Listbox has two molds: default and select. If the select mold is used, the HTML's SELECT tag is generated instead.

```
<listbox mold="select">...</listbox>
```

Notice: if mold is "select", rows is "1", and none of items is marked as selected, the browser displays the listbox as if the first item is selected. Worse of all, if user selects the first item in this case, no onSelect event is sent. To avoid this confusion, developers shall select at least one item for mold="select" and rows="1".

In addition to label, you can assign an application-specific value to each item using the setValue method.

Mouseless Entry

- UP and DOWN to move the selection up and down one list item.
- PgUp and PgDn to move the selection up and down in a step of one page.
- HOME to move the selection to the first item, and END to the last item.
- Ctrl+UP and Ctrl+DOWN to move the focus up and down one list item without changing the selection.
- SPACE to select the item of the focus.

Multi-Column List Boxes

The list box also supports multiple columns. When user selects an item, the entire row is selected.

To specify a multi-column list, you need to specify the listcell components as columns of each listitem (as a row).
<listbox width="200px">
  <listitem>
    <listcell label="George"/>
    <listcell label="House Painter"/>
  </listitem>
  <listitem>
    <listcell label="Mary Ellen"/>
    <listcell label="Candle Maker"/>
  </listitem>
  <listitem>
    <listcell label="Roger"/>
    <listcell label="Swashbuckler"/>
  </listitem>
</listbox>

**Column Headers**

You could specify the column headers by use of `listhead` and `listheader` as follows. In addition to label, you could specify an image as the header by use of the `image` property.

<listbox width="200px">
  <listhead>
    <listheader label="Name"/>
    <listheader label="Occupation"/>
  </listhead>
  ...  
</listbox>

**Column Footers**

You could specify the column footers by use of `listfoot` and `listfooter` as follows. Notice that the order of `listhead` and `listfoot` doesn't matter. Each time a `listhead` instance is added to a list box, it must be the first child, and a `listfoot` instance the last child.

<listbox width="200px">
  <listhead>
    <listheader label="Population"/>
    <listheader align="right" label="%"/>
  </listhead>
  <listitem id="a" value="A">
    | Population | % |
    |-----------|---|
    | A. Graduate | 20% |
    | B. College  | 23% |
    | C. High School | 40% |
    | D. Others   | 17% |
    | More or less | 100% |
  </listitem>
</listbox>

---

4 This feature is a bit different from XUL, where `listhead` and `listheader` are used.
Drop-Down List

You could create a drop-down list by specifying the select mold and single row. Notice you cannot use multi-column for the drop-down list.

```
<listbox mold="select" rows="1">
    <listitem label="Car"/>
    <listitem label="Taxi"/>
    <listitem label="Bus" selected="true"/>
    <listitem label="Train"/>
</listbox>
```

Multiple Selection

When user clicks on a list item, the whole item is selected and the `onSelect` event is sent back to the server to notify the application. You could control whether a list box allows multiple selections by setting the `multiple` property to true. The default value is false.

Scrollable List Boxes

A list box is scrollable if you specify the `rows` property or the `height` property, and there is
The **rows** Property

The **rows** property is used to control how many rows are visible. By setting it to zero, the list box will resize itself to hold as many as items if possible.

**Sizable List Headers**

Like columns, you can set the **sizable** property of **listhead** to true to allow users to resize the width of list headers. Similarly, the **onColResize** event is sent when a user resized the widths.
List Boxes with Paging

Like grids, you can use multiple pages to represent long content for list boxes by specifying the paging mold. Similarly, you can control how many items for each page to display, whether to use an external paging component and whether to customize the behavior when a page is selected. Refer to the Grids section for more details.

Sorting

List boxes support sorting of list items directly. There are a few ways to enable the sorting of a particular column. The simplest way is to set the sort property of the list header to auto as follows. Then, the column that the list header is associated with is sortable based on the label of each list cell of the specified column.

```zk
<listbox width="200px">
  <listhead>
    <listheader label="name" sort="auto"/>
    <listheader label="gender" sort="auto"/>
  </listhead>
  <listitem>
    <listcell label="Mary"/>
    <listcell label="FEMALE"/>
  </listitem>
  <listitem>
    <listcell label="John"/>
    <listcell label="MALE"/>
  </listitem>
  <listitem>
    <listcell label="Jane"/>
    <listcell label="FEMALE"/>
  </listitem>
  <listitem>
    <listcell label="Henry"/>
    <listcell label="MALE"/>
  </listitem>
</listbox>
```

The sortAscending and sortDescending Properties

If you prefer to sort list items in different ways, you can assign a java.util.Comparator
instance to the sortAscending and/or sortDescending property. Once assigned, the list items can be sorted in the ascending and/or descending order with the comparator you assigned.

The invocation of the sort property with auto actually assign two comparators to the sortAscending and sortDescending automatically. You can override any of them by assigning another comparator to it.

For example, assume you want to sort based on the value of list items, rather than list cell's label, then you assign an instance of ListItemComparator to these properties as follows.

```xml
<variable name="asc" value="new ListItemComparator(-1, true, true)"/>
<variable name="dsc" value="new ListItemComparator(-1, false, true)"/>
<listbox>
  <listhead>
    <listheader sortAscending="${asc}" sortDescending="${dsc}"/>
  ...
```

**The sortDirection Property**

The sortDirection property controls whether to show an icon at the client to indicate the order of the particular column. If list items are sorted before adding to the list box, you shall set this property explicitly.

```xml
<listheader sortDirection="ascending"/>
```

Then, it is maintained automatically by list boxes as long as you assign the comparator to the corresponding list header.

**The onSort Event**

When you assign at least one comparator to a list header, an onSort event is sent to the server if user clicks on it. The list header implements a listener to handle the sorting automatically.

If you prefer to handle it manually, you can add your listener to the list header for the onSort event. To prevent the default listener to invoke the sort method, you have to call the stopPropagation method against the event being received. Alternatively, you can override the sort method, see below.

**The sort Method**

The sort method is the underlying implementation of the default onSort event listener. It
is also useful if you want to sort the list items by Java codes. For example, you might have
to call this method after adding items (assuming not in the proper order).

```java
new Listem("New Stuff").setParent(listbox);
if ("natural".header.getSortDirection())
    header.sort("ascending".equals(header.getSortDirection()));
```

The default sorting algorithm is quick-sort (by use of the `sort` method from the
`org.zkoss.zk.ui.Components` class). You might override it with your own
implementation, or listen to the `onSort` event as described in the previous section.

**Tip:** Sorting huge number of live data might degrade the performance significantly. It is
better to intercept the `onSort` event or the `sort` method to handle it effectively. Refer to
the **Sort Live Data** section below.

### Special Properties

#### The checkmark Property

The `checkmark` property controls whether to display a checkbox or a radio button in front
of each list item.

In the following example, you will see how a checkbox is added automatically, when you
move a list item from the left list box
to the right one. The checkbox is
removed when you move a list item
from right to left.

```
<listbox id="src" rows="0" multiple="true" width="200px">
    <listhead>
        <listheader label="Population"/>
        <listheader label="Percentage"/>
    </listhead>
    <listitem id="a" value="A">
        <listcell label="A. Graduate"/>
        <listcell label="20\%"/>
    </listitem>
    <listitem id="b" value="B">
        <listcell label="B. College"/>
        <listcell label="23\%"/>
    </listitem>
    <listitem id="c" value="C">
```

---

ZK: Developer’s Guide Page 60 of 92 Potix Corporation
Notice that if the `multiple` property is false, the radio buttons are displayed instead, as depicted at the right.

**The `vflex` Property**

The `vflex` property controls whether to grow and shrink vertical to fit their given space.
It is so-called vertical flexibility. For example, if the list is too big to fit in the browser window, it will shrink its height to make the whole list control visible in the browser window.

This property is ignored if the rows property is specified.

The maxlength Property

The maxlength property defines the maximal allowed characters being visible at the browser. By setting this property, you could make a narrower list box.

List Boxes Contain Buttons

In theory, a list cell could contain any other components, as depicted below.

```html
<listbox width="250px">
  <listhead>
    <listheader label="Population"/>
    <listheader label="Percentage"/>
  </listhead>
  <listitem value="A">
    <listcell><textbox value="A. Graduate"/></listcell>
    <listcell label="20%"/>
  </listitem>
  <listitem value="B">
    <listcell><checkbox label="B. College"/></listcell>
    <listcell><button label="23%"/></listcell>
  </listitem>
  <listitem value="C">
    <listcell label="C. High School"/>
    <listcell><textbox cols="8" value="40%"/></listcell>
  </listitem>
</listbox>
```

Notes:

1. Don't use a list box, when a grid is a better choice. The appearances of list boxes and grids are similar, but the list box shall be used only to represent a list of selectable items.

2. Users are usually confused if a list box contains editable components, such as textbox and checkbox. A common question is what the text, that a user entered in a unselected item, means.

3. Due to the limitation of the browsers, users cannot select a piece of characters from the text boxes.
Tree Controls

Components: tree, treechildren, treeitem, treerow, treecell, treecols and treecol.

A tree consists of two parts, the set of columns, and the tree body. The set of columns is defined by a number of treecol components, one for each column. Each column will appear as a header at the top of the tree. The second part, the tree body, contains the data to appear in the tree and is created with a treechildren component.

An example of a tree control is as follows.

```
<tree id="tree" rows="5">
  <treecols>
    <treecol label="Name"/>
    <treecol label="Description"/>
  </treecols>
  <treechildren>
    <treeitem>
      <treerow>
        <treecell label="Item 1"/>
        <treecell label="Item 1 description"/>
      </treerow>
    </treeitem>
    <treeitem>
      <treerow>
        <treecell label="Item 2"/>
        <treecell label="Item 2 description"/>
      </treerow>
    </treeitem>
    <treechildren>
      <treeitem>
        <treerow>
          <treecell label="Item 2.1"/>
        </treerow>
      </treeitem>
      <treeitem>
        <treerow>
          <treecell label="Item 2.1.1"/>
        </treerow>
      </treeitem>
      <treeitem>
        <treerow>
          <treecell label="Item 2.1.2"/>
        </treerow>
      </treeitem>
    </treechildren>
  </treechildren>
</tree>
```
• tree: This is the outer component of a tree control.

• treecols: This component is a placeholder for a collection of treecol components.

• treecol: This is used to declare a column of the tree. By using this column, you can specify additional information such as the column header.

• treechildren: This contains the main body of the tree, which contain a collection of treeitem components.

• treeitem: This component contains a row of data (treerow), and an optional treechildren.
  o If the component doesn't contain a treechildren, it is a leaf node that doesn't accept any child items.
  o If it contains a treechildren, it is a branch node that might contain other items.
  o For a branch node, an +/- button will appear at the beginning of the row, such that user could open and close the item by clicking on the +/- button.

• treerow: A single row in the tree, which should be placed inside a treeitem component.

• treecol: A single cell in a tree row. This element would go inside a treerow component.

<table>
<thead>
<tr>
<th>Mouseless Entry</th>
<th>tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP and DOWN</td>
<td></td>
</tr>
<tr>
<td>PgUp and PgDn</td>
<td></td>
</tr>
<tr>
<td>HOME</td>
<td></td>
</tr>
</tbody>
</table>

• UP and DOWN to move the selection up and down one tree item.
• PgUp and PgDn to move the selection up and down in a step of one page.
• HOME to move the selection to the first item, and END to the last item.
- RIGHT to open a tree item, and LEFT to close a tree item.
- Ctrl+UP and Ctrl+DOWN to move the focus up and down one tree item without changing the selection.
- SPACE to select the item of the focus.

**The open Property and the onOpen Event**

Each tree item has the open property used to control whether to display its child items. The default value is true. By setting this property to false, you could control what part of the tree is invisible.

```xml
<treeitem open="false"/>
```

When a user clicks on the +/- button, he opens the tree item and makes its children visible. The onOpen event is then sent to the server to notify the application.

For sophisticated applications, you can defer the creation of the content of the tree item or manipulate its content dynamically, until the onOpen event is received. Refer to the Load on Demand section in the ZK User Interface Markup Language chapter for details.

**Multiple Selection**

When user clicks on a tree item, the whole item is selected and the onSelect event is sent back to the server to notify the application. You could control whether a tree control allows multiple selections by setting the multiple property to true. The default value is false.

**Paging**

The pageSize property controls the number of tree items to display at once. By default, it is 10. That is, at most 10 tree items are displayed at the client for each level as depicted in the right figure.

A user can click ⊹ to see more tree items (i.e., enlarge pageSize), or click ▲ or ▼ to scroll up and down.

If you want to display all tree items, simply set pageSize to -1. However, it is not recommended if the tree control is huge, since the browser is too slow to handle a tree with huge number of items.

In addition to the pageSize property of a tree control, you can change the page size of each treechildren instance by modifying the pageSize property of the corresponding treechildren instance.
**The `onPaging` and `onPageSize` Event**

When a user clicks ▲ or ▼ to scroll up and down the page, the `onPaging` event is sent with an `org.zkoss.zul.event.PagingEvent` instance. Similarly, the `onPageSize` event is sent with an `org.zkoss.zul.event.PageSize` instance when a user clicks ⌘.

**Special Properties**

**The `rows` Property**

The `rows` property is used to control how many rows are visible. By setting it to zero, the tree control will resize itself to hold as many as items if possible.

**The `checkmark` Property**

The `checkmark` property controls whether to display a checkbox or a radio button in front of each tree item.

**The `vflex` Property**

The `vflex` property controls whether to grow and shrink vertical to fit their given space. It is so-called vertical flexibility. For example, if the tree is too big to fit in the browser window, it will shrink the height to make the whole tree visible in the browser window.

This property is ignored if the `rows` property is specified.

**The `maxlength` Property**

The `maxlength` property defines the maximal allowed characters being visible at the browser. By setting this property, you could make a narrower tree control.

**Sizable Columns**

Like `columns`, you can set the `sizable` property of `treecols` to `true` to allow users to resize the width of tree headers. Similarly, the `onColSize` event is sent when a user
Create-on-Open for Tree Controls

As illustrated below, you could listen to the **onOpen** event, and then load the children of an tree item. Similarly, you could do create-on-open for group boxes.

```xml
<tree width="200px">
  <treecols>
    <treecol label="Subject"/>
    <treecol label="From"/>
  </treecols>
  <treechildren>
    <treeitem open="false">
      <attribute name="onOpen">
        Treechildren tc = self.getTreechildren();
        if (tc.getChildren().isEmpty()) {
          Treeitem ti = new Treeitem();
          ti.setLabel("New added");
          ti.setParent(tc);
        }
      </attribute>
      <treerow>
        <treecell label="Intel Snares XML"/>
        <treecell label="David Needle"/>
      </treerow>
    </treeitem>
  </treechildren>
</tree>
```

Comboboxes

**Components:** `combobox` and `comboitem`.

A combobox is a special text box that embeds a drop-down list. With comboboxes, users are allowed to select from a drop-down list, in addition to entering the text manually.

```xml
<combobox>
  <comboitem label="Simple and Rich"/>
  <comboitem label="Cool!">
</combobox>
```
The autodrop Property

By default, the drop-down list won't be opened until user clicks the button, or press Alt+DOWN. However, you could set the autodrop property to true, such that the drop-down list is opened as soon as user types any character. This is helpful for novice users, but it might be annoying for experienced users.

```html
<combobox autodrop="true"/>
```

The description Property

You could add a description to each combo item to make it more descriptive. In addition, you could assign an image to each combo item.

```html
<combobox>
  <comboitem label="Simple and Rich" image="/img/coffee.gif" description="The simplest way to make Web applications rich"/>
  <comboitem label="Cool!" image="/img/cool.gif" description="The coolest technology"/>
  <comboitem label="Ajax and RIA" image="/img/cubfirs.gif" description="Rich Internet Application by Ajax"/>
</combobox>
```

Like other components that support images, you could use the setImageContent method to assign the content of a dynamically generated image to the comboitem component. Refer to the Image section for details.

The onOpen Event

The onOpen event is sent to the application, when user opens the drop-down list. To defer the creation of combo items, you can use the fulfill attribute as shown below.

```html
<combobox fulfill="onOpen">
  <comboitem label="Simple and Rich"/>
</combobox>
```
Alternatively, you can listen to the `onOpen` event, and then prepare the drop-down list or change it dynamically in the listener as shown below.

```xml
<combobox id="combo"><attribute name="onOpen="">
    if (event.isOpen() && combo.getItemCount() == 0) {
        combo.appendItem("Simple and Rich");
        combo.appendItem("Cool!");
        combo.appendItem("Ajax and RIA");
    }
</attribute></combobox>
```

The `appendItem` method is equivalent to create a combo item and then assign its parent to the combobox.

**The `onChanging` Event**

Since a combobox is also a text box, the `onChanging` event will be sent if you add a listener for it. By listening to this event, you could manipulate the drop-down list as the Google Suggests\(^5\) does. This feature is sometimes called autocomplete.

As illustrated below, you could fill the drop-down list based on what user is entering.

```xml
<combobox id="combo" autodrop="true"><attribute name="onChanging" >
    combo.getItems().clear();
    if (event.value.startsWith("A")) {
        combo.appendItem("Ace");
        combo.appendItem("Ajax");
        combo.appendItem("Apple");
    } else if (event.value.startsWith("B")) {
        combo.appendItem("Best");
        combo.appendItem("Blog");
    }
</attribute></combobox>
```

Notice that, when the `onChanging` event is received, the content of the combobox is not changed yet. Thus, you cannot use the `value` property of the combobox. Rather, you shall use the `value` property of the event (`org.zkoss.zk.ui.event.InputEvent`).

---

\(^5\) [http://www.google.com/webhp?complete=1&hl=en](http://www.google.com/webhp?complete=1&hl=en)
Bandboxes

Components: **bandbox** and **bandpopup**.

A bandbox is a special text box that embeds a customizable popup window (aka., a dropdown window). Like combobxes, a bandbox consists of an input box and a popup window. The popup window is opened automatically, when users presses **Alt+DOWN** or clicks the ![button](image). Unlike combobxes, the popup window of a bandbox could be anything. It is designed to give developers the maximal flexibility. A typical use is to represent the popup window as a search dialog.

```xml
<BANDBOX id="bd">
  <BANDPOPUP>
    <VBOX>
      <HBOX>Search <TEXTBOX/></HBOX>
      <LISTBOX width="200px" onselect="bd.value=self.selectedItem.label; bd.closeDropdown();">
        <LISTHEAD>
          <LISTHEADER label="Name"/>
          <LISTHEADER label="Description"/>
        </LISTHEAD>
        <LISTITEM>
          <LISTCELL label="John"/>
          <LISTCELL label="CEO"/>
        </LISTITEM>
        <LISTITEM>
          <LISTCELL label="Joe"/>
          <LISTCELL label="Engineer"/>
        </LISTITEM>
        <LISTITEM>
          <LISTCELL label="Mary"/>
          <LISTCELL label="Supervisor"/>
        </LISTITEM>
      </LISTBOX>
    </VBOX>
  </BANDPOPUP>
</BANDBOX>
```

**Mouseless Entry**

- **Alt+DOWN** to pop up the list.
- **Alt+UP** or **ESC** to close the list.
- **UP** and **DOWN** to change the selection of the items from the list.

**The closeDropdown Method**

A popup window could contain any kind of components, so it is developer's job to copy the
value from and close the popup if one of item is selected.

In the above example, we copy the selected item’s label to the bandbox, and then close the popup by the following statement.

```html
<listbox width="200px"
  onSelect="bd.value=self.selectedItem.label; bd.closeDropdown();">
</listbox>
```

**The autodrop Property**

By default, the popup window won’t be opened until user clicks the button, or press Alt+DOWN. However, you could set the autodrop property to true, such that the popup is opened as soon as user types any character. This is helpful for novice users, but it might be annoying for experienced users.

```html
<bandbox autodrop="true"/>
```

**The onOpen Event**

The onOpen event is sent to the application if the user opens the popup window. By use of the fulfill attribute with the onOpen value as shown below, you can defer the creation of the popup window.

```html
<bandbox fulfill="onOpen">
  <bandpopup>
    ...
  </bandpopup>
</bandbox>
```

Alternatively, you could prepare the popup window in Java by listening to the onOpen event, as depicted below.

```java
<bandbox id="band">
  <attribute name="onOpen">
    if (event.isOpen() && band.getPopup() == null) {
      ...//create child elements
    }
  </attribute>
</bandbox>
```

**The onChanging Event**

Since a bandbox is also a text box, the onChanging event will be sent if you add a listener for it. By listening to this event, you could manipulate the popup window any way you like.

As illustrated below, you could fill the drop-down list based on what user is entering.
Notice that, when the `onChanging` event is received, the content of the bandbox is not changed yet. Thus, you cannot use the `value` property of the bandbox. Rather, you shall use the `value` property of the event (`org.zkoss.zk.ui.event.InputEvent`).

**Chart**

**Components:** chart

A chart is used to show a set of data as a graph. It helps users to judge things with a snapshot.

The usage of chart component is straightforward. Prepare suitable data model and feed it into the chart. The following is an example of pie chart.

```java
// In Java somewhere:
PieModel pieModel = new SimplePieModel();
model.setValue("C/C++", new Double(17.5));
model.setValue("PHP", new Double(32.5));
model.setValue("Java", new Double(43.2));
model.setValue("VB", new Double(10.0));
```

```zhtml
<chart id="mychart" type="pie" width="400" height="200" threeD="true" fgAlpha="128"
model="${pieModel}"/>
```

Different kind of chart is used to demonstrate different kind of data; therefore, chart has to be provided suitable data model. For a pie chart, developers must provide PieModel as their data model while bar chart, line chart, area chart, and waterfall chart needs CategoryModel and XYModel.

**Live Data**

The above example is somehow a little bit misleading. In fact, developers don’t have to
prepare the real data before feed it into a chart because chart components support live data mechanism. With live data, developers could separate the data from the view. In other words, the developer can add, change, and remove data from the data model and the chart would be redrawn accordingly. For some advanced implementation, developers can even provide their own chart model by implementing the org.zkoss.zul.ChartModel interface.

**Drill Down (The onClick Event)**

When a user views a chart and finds something interesting, it is natural that the user would like to see the detail information regarding that interesting point. It is usually a pie in a pie chart, a bar in a bar chart or a point in a line chart. Chart components support such drill down facility by automatically cutting a chart into area components and users can then click on the chart to fire an onClick MouseEvent. Developers then locate the area component and do whatever appropriate drill down.

In the area component's componentScope there are some useful information that developers can use them.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>Entity type of the area. (e.g. TITLE, DATA, CATEGORY, LEGEND)</td>
</tr>
<tr>
<td>Series</td>
<td>Series name of the associated data (CategoryModel, XYModel, or HiLoModel).</td>
</tr>
<tr>
<td>Category</td>
<td>Category name of the associated data (PieModel or CategoryModel).</td>
</tr>
<tr>
<td>url</td>
<td>An url in string that can be used to drill down to a legacy page.</td>
</tr>
<tr>
<td>Value</td>
<td>Numeric value of the associated data (PieModel or CategoryModel).</td>
</tr>
<tr>
<td>X</td>
<td>x value of the associated data (XYModel).</td>
</tr>
<tr>
<td>Y</td>
<td>y value of the associated data (XYModel).</td>
</tr>
<tr>
<td>Date</td>
<td>date value of the associated data (HiLoModel).</td>
</tr>
<tr>
<td>Open</td>
<td>open value of the associated data (HiLoModel).</td>
</tr>
<tr>
<td>High</td>
<td>high value of the associated data (HiLoModel).</td>
</tr>
<tr>
<td>Low</td>
<td>low value of the associated data (HiLoModel).</td>
</tr>
<tr>
<td>Close</td>
<td>close value of the associated data (HiLoModel).</td>
</tr>
<tr>
<td>Volume</td>
<td>volume value of the associated data (HiLoModel).</td>
</tr>
</tbody>
</table>

In the following example, we provide an onClick event listener on the chart. It locates the associated area component and show the category of that area (i.e. the pie).

```xml
<chart id="mychart" type="pie" width="400" height="250" model="${pieModel}"
<attribute name="onClick">
```
Manipulate Areas

Chart components also provide a area renderer mechanism that developers can manipulate the cutting area components of the chart.

Only two steps needed to use the area renderer.

1. Implement the `org.zkoss.zul.event.ChartAreaListener` interface for manipulating the area components. e.g. Change the `tooltiptext` of the area.

2. Set the listener object or listener class name to the chart's `areaListener` property.

So developers get a chance to change the area component's properties or insert more information into the area component's `componentScope` property and thus be passed through to the `onClick` event listener.

Drag and Drop

ZK allows a user to drag particular components around within the user interface. For example, dragging files to other directories, or dragging an item to the shopping cart to purchase.

A component is draggable if it can be dragged around. A component is droppable, if a user could drop a draggable component to it.

**Note:** ZK does not assume any behavior about what shall take place after dropping. It is up to application developers by writing the `onDrop` event listener.

If an application doesn't nothing, the dragged component is simply moved back where it is originated from.

The draggable and droppable Properties

With ZK, you could make a component draggable by assigning any value, other than "false", to the draggable property. To disable it, assign it with "false".

```xml
<image draggable="true"/>
```

Similarly, you could make a component droppable by assigning "true" to the droppable property.

```xml
<hbox droppable="true"/>
```
Then, user could drag a draggable component, and then drop it to a droppable component.

**The onDrop Event**

Once user has dragged a component and dropped it to another component, the component that the user dropped the component to will be notified by the onDrop event. The onDrop event is an instance of org.zkoss.ui.event.DropEvent. By calling the getDragged method, you could retrieve what has been dragged (and dropped).

Notice that the target of the onDrop event is the droppable component, not the component being dragged.

The following is a simple example that allows users to reorder list items by drag-and-drop.

```xml
<window title="Reorder by Drag-and-Drop" border="normal">
  Unique Visitors of ZK:
  <listbox id="src" multiple="true" width="300px">
    <listhead>
      <listheader label="Country/Area"/>
      <listheader align="right" label="Visits"/>
      <listheader align="right" label="%"/>
    </listhead>
    <listitem draggable="true" droppable="true"><attribute name="onDrop">
      self.parent.insertBefore(event.dragged, self);
    </attribute>
      <listcell label="United States"/>
      <listcell label="5,093"/>
      <listcell label="19.39%"/>
    </listitem>
    <listitem draggable="true" droppable="true"><attribute name="onDrop">
      self.parent.insertBefore(event.dragged, self);
    </attribute>
      <listcell label="China"/>
      <listcell label="4,274"/>
      <listcell label="16.27%"/>
    </listitem>
    <listitem draggable="true" droppable="true"><attribute name="onDrop">
      self.parent.insertBefore(event.dragged, self);
    </attribute>
      <listcell label="France"/>
      <listcell label="1,892"/>
      <listcell label="7.20%"/>
    </listitem>
    <listitem draggable="true" droppable="true"><attribute name="onDrop">
      self.parent.insertBefore(event.dragged, self);
    </attribute>
      <listcell label="Germany"/>
      <listcell label="1,866"/>
      <listcell label="7.03%"/>
    </listitem>
    <listitem draggable="true" droppable="true"><attribute name="onDrop">
      self.parent.insertBefore(event.dragged, self);
    </attribute>
      <listcell label="(other) - France"/>
      <listcell label="1,892"/>
      <listcell label="7.20%"/>
    </listitem>
    <listitem draggable="true" droppable="true"><attribute name="onDrop">
      self.parent.insertBefore(event.dragged, self);
    </attribute>
      <listcell label="Total 132"/>
      <listcell label="26,287"/>
      <listcell label="100.00%"/>
  </listbox>
</window>
```
Dragging with Multiple Selections

When a user drag-and-drops a list item or a tree item, the selection status of these items won’t be changed. Visually only the dragged item is moved, but you can handle all selected items at once by looking up the set of all selected items as depicted below.

```java
public void onDrop(DropEvent evt) {
    Set selected = ((Listitem)evt.getDragged()).getListbox().getSelectedItems();
    // then, you can handle the whole set at once
}
```

Notice that the dragged item may not be selected. Thus, you may prefer to change the selection to the dragged item for this case, as shown below.

```java
Listitem li = (Listitem)evt.getDragged();
if (li.isSelected()){
    Set selected = ((Listitem)evt.getDragged()).getListbox().getSelectedItems();
    // then, you can handle the whole set at once
} else {
```
Multiple Types of Draggable Components

It is common that a draggable component doesn’t accept all draggable components. For example, an e-mail folder accepts only e-mails and it rejects contacts or others. You could silently ignore non-acceptable components or alert an message, when `onDrop` is invoked.

To have better visual effect, you could identify each type of draggable components with an identifier, and then assign the identifier to the `draggable` property.

```html
<li draggable="email"/>  
...  
<li draggable="contact"/>
```

Then, you could specify a list of identifiers to the `droppable` property to limit what can be dropped. For example, the following image accepts only email and contact.

```html
<image src="/img/send.png" droppable="email, contact" onDrop="send(event.dragged)"/>
```

To accept any kind of draggable components, you could specify "true" to the `droppable` property. For example, the following image accepts any kind of draggable components.

```html
<image src="/img/trash.png" droppable="true" onDrop="remove(event.dragged)"/>
```

On the other hand, if the `draggable` property is "true", it means the component belongs to anonymous type. Furthermore, only components with the `droppable` property assigned to "true" could accept it.

HTML Relevant Components

There are several ways to use HTML components with XUL components in the same ZUML page.

The `style` Component

The `style` component is used to specify CSS styles in a ZUML page. The simplest format is as follows.

```html
<style>  
  .blue {  
    color: blue;  
  }
</style>
```
Tip: To configure a style sheet for the whole application, specify theme-uri in zk.xml, refer to Appendix B in the Developer's Reference for details. To configure a style sheet for a language, use the language addon, refer to the Component Development Guide.

Sometimes it is better to store all CSS definitions in an independent file, say my.css. Then, we could reference it by use of the style component as follows.

```html
<style src="/my.css"/>
```

The above statement actually sends the following HTML tags\(^6\) to the browser, so the specified file must be accessible by the browser.

```html
<link rel="stylesheet" href="/css/mystyles.css"/>
```

In other words, you cannot specify "/WEB-INF/xx" or "C:/xx/yy".

Like other URI, it accepts "*" for loading browser and Locale dependent style sheet. Refer to the Browser and Locale Dependent URI section in the Internationalization chapter for details.

The **include Component**

The **include** component is used to include the output generated by another servlet. The servlet could be anything including JSF, JSP and even another ZHTML page.

```html
<window title="include demo" border="normal" width="300px">

Hello, World!

<include src="/userguide/misc/includedHello.zhtml"/>
<include src="/html/frag.html"/>

</window>
```

Like all other properties, you could dynamically change the **src** attribute to include the output from a different servlet at the run time.

If the included output is another ZUML, developers are allowed to access components in the included page as if they are part of the containing page.

\(^6\) The real result depends on how your Web application is configured.
Including ZHTML Pages

If the `include` component is used to include a ZUML page, the included page will become part of the desktop. However, the included page is not visible until the request is processed completely. In other words, it is visible only in the following events, triggered by user or timer.

The `iframe` Component

The `iframe` component uses the HTML IFRAME tag to delegate a portion of the display to another URL. Though the appearance looks similar to the `include` component. The concept and meaning of the `iframe` component is different.

The content included by the `include` component is a fragment of the whole HTML page. Because the content is part of the HTML page, the content is part of the desktop and you could access any components, if any, inside of the `include` component. The inclusion is done at the server, and the browser knows nothing about it. It means the URL specified by the `src` property could be any internal resource.

The content of the `iframe` component is loaded by the browser as a separate page. Because it is loaded as a separate page, the format of the content could be different from HTML. For example, you could embed an PDF file.

```html
<iframe src="/my.pdf"/>
...other HTML content
```

**Tip:** By default, there is no border. To enable it, use the style attribute to specify it. For example,

```html
<iframe style="border:1px inset" src="http://www.zkoss.org"/>
```

The *embedding* is done by the browser, when it interprets the HTML page containing the IFRAME tag. It also implies that the URL must be a resource that you can access from the browser.

Like the `image` and `audio` components\(^7\), you could specify the dynamically generated content. A typical example is you could use JasperReport\(^8\) to generate a PDF report in a binary array or stream, and then pass the report to an `iframe` component by wrapping the result with the `org.zkoss.util.media.AMedia` class.

In the following example, we illustrate that you could embed any content by use of `iframe`, as long as the client supports its format.

---

\(^7\) In many ways, `iframe` is much similar to `image` and `audio`. You might consider it as a component for arbitrary content.

\(^8\) [http://jasperreports.sourceforge.net](http://jasperreports.sourceforge.net)
### Client Side Actions

Some behaviors are better to be done at the client side with JavaScript codes, such as animations and image rollovers. In order to execute JavaScript codes at the client, ZK introduces the concept of Client Side Actions (CSA). With CSA, developers could listen to any JavaScript event and executes JavaScript codes at the client.

A CSA is similar to an event listener, except an action is is written in JavaScript and executes at the client. ZK allows developers to specify actions for any JavaScript events, such as `onfocus`, `onblur`, `onmouseover` and `onmouseout`, as long as your targeting browsers support them.

The syntax of a client-side action is as follows.

```
action="[onfocus|onblur|onmouseover|onmouseout|onclick|onshow|onhide...]: javascript;"
```

Notice that CSA is totally independent of ZK event listeners, though they might have the same name, such as `onFocus`. The differences include:

- CSA executes at the client side and takes place, before ZK event listener is called at the server.
- CSA codes are written in JavaScript, while ZK event listeners are written in Java.
- CSA could register to any event that your targeting browsers allow, while ZK supports events only list in the **Events** section.

**Reference to a Component**

In the JavaScript codes, you can reference to a component or other objects with the late-binding EL expression. The late-binding EL expression starts with `#{` and ending with `}` as depicted below.

```html
<button action="onmouseover: action.show(#{parent.tip})"/>
```

The late-binding EL expressions are evaluated as late as the Rendering Phase. On the other hand, if you assign an EL expression starting with `${`, it will be evaluated at the Component Creation Phase, before assigning to the `action` property. For example,

```html
<button action="onfocus: action.show(${tip}); onblur: action.hide(${tip})"/>
<div id="tip" visible="false">...</div>
```

will be evaluated to

```html
<button action="onfocus: action.show(); onblur: action.hide()"/>
<div id="tip" visible="false">...</div>
```

since the `tip` component is not created when assigning the `action` property.

Even if the referenced component was created before `action` is assigned, it is still incorrect, since the ZUML loader has no knowledge of CSA, and it converts the component to a string by invoking the `toString` method.

Of course, it doesn't prevent you from using `{} in an action, as depicted below. Just remember it is evaluated before assigning the `action` property.

```html
<variables myaction="onfocus: action.show(${tip}); onblur: action.hide(${tip});;"
<button action="${myaction} onmouseover: action.show(#{parent.parent.tip})"/>
```

**An onfocus and onblur Example**

In the following example, we demonstrated how to use client-side actions to provide on-line help. When user change the focus to any of the text boxes, a help message is displayed accordingly.

```html
<grid>
  <columns>
    <column/>
  </column/>
  text1: [This is help for text1.]
  text2: [This is help for text2.]
</grid>
```
Coercion Rules

A ZUL component actually converts an EL expression (#{}) to proper JavaScript codes based on the class of the result object.

1. If result is null, it is replaced with null.
2. If result is a component, it is replaced with $e('uuid'), where $e is a JavaScript function to return a reference to a HTML tag and uuid is the component's UUID.
3. If result is a Date object, it is replaced with new Date(milliseconds).
4. Otherwise, the result is converted to a string by calling the toString method, and then replaced with 'result in string'.

The onshow and onhide Actions

The onshow and onhide actions are used to control the visual effect of displaying and hiding a component.

An Example to Change How a Window Appears

```<zk>
  <button label="Show Overlapped"><attribute name="onClick">
    win.doOverlapped();
  </attribute>
</zk>```
CSA JavaScript Utilities

To simplify the CSA programming, ZK provides a few utilities objects that you can utilize.

The action Object

Basic utilities that can be applied to any object.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action.show(cmp)</td>
<td>Make a component visible.</td>
</tr>
<tr>
<td></td>
<td>cmp – the component. Use #{EL-expr} to identify it.</td>
</tr>
<tr>
<td>action.hide(cmp)</td>
<td>Make a component invisible.</td>
</tr>
<tr>
<td></td>
<td>cmp – the component. Use #{EL-expr} to identify it.</td>
</tr>
</tbody>
</table>

**Tip:** For JavaScript programmers, it is common to manipulate style.display directly for visibility. However, it is not a good idea. Rather, use action.show and action.hide instead, since ZK Client Engine has to handle visual effects, bug workaround, and so on.

The anima Object

Animation-like visual effects. It is based on the Effect class provided by script.aculo.us. The API is simplified. If you’d like more visual effects or controls, you can access Effect directly.

**Note:** Effect requires the component to be enclosed with the DIV tag. Not all ZUL components are implemented in this way. If you have any doubt, you can nest it with the div component as follows.

```html
<window>
  <div id="t" visible="false"
    action="onshow: anima.slideDown({self}); onhide: anima.slideUp({self})">
```

---

http://script.aculo.us provides easy-to-use, cross-browser user interface JavaScript libraries
Of course, you load other libraries that do not have this limitation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anima.appear(cmp)</td>
<td>Make a component visible by increasing the opacity.</td>
</tr>
<tr>
<td>anima.appear(cmp, dur)</td>
<td>- cmp – the component. Use ${EL-expr} to identify it.</td>
</tr>
<tr>
<td></td>
<td>- dur – the duration in milliseconds. Default: 800.</td>
</tr>
<tr>
<td>anima.slideDown(cmp)</td>
<td>Make a component visible with the slide-down effect.</td>
</tr>
<tr>
<td>anima.slideDown(cmp, dur)</td>
<td>- cmp – the component. Use ${EL-expr} to identify it.</td>
</tr>
<tr>
<td></td>
<td>- dur – the duration in milliseconds. Default: 400.</td>
</tr>
<tr>
<td>anima.slideUp(cmp)</td>
<td>Make a component invisible with the slide-up effect.</td>
</tr>
<tr>
<td>anima.slideUp(cmp, dur)</td>
<td>- cmp – the component. Use ${EL-expr} to identify it.</td>
</tr>
<tr>
<td></td>
<td>- dur – the duration in milliseconds. Default: 400.</td>
</tr>
<tr>
<td>anima.fade(cmp)</td>
<td>Make a component invisible by fading it out.</td>
</tr>
<tr>
<td>anima.fade(cmp, dur)</td>
<td>- cmp – the component. Use ${EL-expr} to identify it.</td>
</tr>
<tr>
<td>anima.puff(cmp)</td>
<td>Make a component invisible by puffing it out.</td>
</tr>
<tr>
<td>anima.puff(cmp, dur)</td>
<td>- cmp – the component. Use ${EL-expr} to identify it.</td>
</tr>
<tr>
<td></td>
<td>- dur – the duration in milliseconds. Default: 700.</td>
</tr>
<tr>
<td>anima.dropOut(cmp)</td>
<td>Make a component invisible by fading and dropping it.</td>
</tr>
<tr>
<td>anima.dropOut(cmp, dur)</td>
<td>- cmp – the component. Use ${EL-expr} to identify it.</td>
</tr>
<tr>
<td></td>
<td>- dur – the duration in milliseconds. Default: 700.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>out.</td>
<td></td>
</tr>
<tr>
<td>cmp – the component. Use #{EL-expr} to identify it.</td>
<td></td>
</tr>
<tr>
<td>dur – the duration in milliseconds. Default: 700.</td>
<td></td>
</tr>
</tbody>
</table>

For example,

```html
<window title="Animation Effects">
  <style>
    .ctl {
      border: 1px outset #777; background:#ddeecc;
      margin: 2px; margin-right: 10px; padding-left: 2px; padding-right: 2px;
    }
  </style>

  <label value="Slide" sclass="ctl"
    action="onmouseover: anima.slideDown(#\{t\}); onmouseout: anima.slideUp(#\{t\})"/>
  <label value="Fade" sclass="ctl"
    action="onmouseover: anima.appear(#\{t\}); onmouseout: anima.fade(#\{t\})"/>
  <label value="Puff" sclass="ctl"
    action="onmouseover: anima.appear(#\{t\}); onmouseout: anima.puff(#\{t\})"/>
  <label value="Drop Out" sclass="ctl"
    action="onmouseover: anima.appear(#\{t\}); onmouseout: anima.dropOut(#\{t\})"/>

  <div id="t" visible="false">
    <div>
      <groupbox>
        <caption label="Dynamic Content"/>
        Content to show and hide dynamically.
        <datebox/>
      </groupbox>
      Description <textbox/>
    </div>
  </div>
</window>
```

**Events**

Notice that whether an event is supported depends on a component. In addition, an event is sent after the component's content is updated.

**Mouse Events**

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Components/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Name</td>
<td>Components/Description</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>onClick</td>
<td>button caption column div groupbox image imagemap label listcell listfooter listheader menuitem tabpanel toolbar toolbarbutton treecell treecol window</td>
</tr>
<tr>
<td>Event: org.zkoss.zk.ui.event.MouseEvent</td>
<td></td>
</tr>
<tr>
<td>Denotes user has clicked the component.</td>
<td></td>
</tr>
<tr>
<td>onRightClick</td>
<td>button caption checkbox column div groupbox image imagemap label listcell listfooter listheader listitem radio slider tab tabbox tabpanel toolbar toolbarbutton treecell treecol treeitem window</td>
</tr>
<tr>
<td>Event: org.zkoss.zk.ui.event.MouseEvent</td>
<td></td>
</tr>
<tr>
<td>Denotes user has right-clicked the component.</td>
<td></td>
</tr>
<tr>
<td>onDoubleClick</td>
<td>button caption checkbox column div groupbox image label listcell listfooter listheader listitem tab tabpanel toolbar toolbarbutton treecell treecol treeitem window</td>
</tr>
<tr>
<td>Event: org.zkoss.zk.ui.event.MouseEvent</td>
<td></td>
</tr>
<tr>
<td>Denotes user has double-clicked the component.</td>
<td></td>
</tr>
</tbody>
</table>

**Keystroke Events**

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onOK</td>
<td>window</td>
<td>Event: org.zkoss.zk.ui.event.KeyEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Denotes user has pressed the <strong>ENTER</strong> key.</td>
</tr>
<tr>
<td>onCancel</td>
<td>window</td>
<td>Event: org.zkoss.zk.ui.event.KeyEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Denotes user has pressed the <strong>ESC</strong> key.</td>
</tr>
<tr>
<td>onCtrl1Key</td>
<td>window</td>
<td>Event: org.zkoss.zk.ui.event.KeyEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Denotes user has pressed a special key, such as <strong>PgUp</strong>, <strong>Home</strong> and a key combined with the <strong>Ctrl</strong> or <strong>Alt</strong> key. Refer to the ctrlKeys Property section below for details.</td>
</tr>
</tbody>
</table>

The keystroke events are sent to the nearest window that has registered an event listener for the specified events. It is designed to implement the submit, cancel and shortcut functions.
As illustrated below, doA() is invoked if user pressed ENTER when T1 got the focus, and doB() is invoked if user pressed ENTER when T2 got the focus.

```xml
<window id="A" onOK="doA()">
  <window id="B" onOK="doB()">
    <textbox id="T1"/>
  </window>
  <textbox id="T2"/>
</window>
```

Notice that a window doesn't receive the keystroke events that are sent for the inner window, unless you post them manually. In the above example, the event won't be sent to window A, if T1 got the focus, no matter whether the onOK handler is declared for window B or not.

**The ctrlKeys Property**

To receive the onCtrlKey event, you must specify what key strokes to intercept by the ctrlKeys property. In other words, only key strokes specified in the ctrlKeys property is sent back to the server. For example, the onCtrlKey event is sent if a user clicks Alt+C, Ctrl+A, F10, or Ctrl+F3.

```xml
<window ctrlKeys="@c^a#10^#3">
  ...
```

The following is the syntax of the ctrlKeys property.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^k</td>
<td>A control key, i.e., Ctrl+k, where k could be a<del>z, 0</del>9, #n and ~n.</td>
</tr>
<tr>
<td>@k</td>
<td>A alt key, i.e., Alt+k, where k could be a<del>z, 0</del>9, #n and ~n.</td>
</tr>
<tr>
<td>$k</td>
<td>A shift key, i.e., Shift+k, where k could be #n and ~n.</td>
</tr>
<tr>
<td>#n</td>
<td>A special key as follows.</td>
</tr>
<tr>
<td></td>
<td>#home Home</td>
</tr>
<tr>
<td></td>
<td>#del Delete</td>
</tr>
<tr>
<td></td>
<td>#up ↑</td>
</tr>
<tr>
<td></td>
<td>#pgdn PgDn</td>
</tr>
<tr>
<td></td>
<td>#fn A function key. #f1, #f2, ... #f12 for F1, F2, ... F12.</td>
</tr>
</tbody>
</table>
## Input Events

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onChange</td>
<td>textbox</td>
<td>Event: org.zkoss.zk.ui.event.InputEvent</td>
</tr>
<tr>
<td></td>
<td>datebox</td>
<td>Denotes the content of an input component has been modified by the user.</td>
</tr>
<tr>
<td></td>
<td>decimalbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>doublebox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>combobox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bandbox</td>
<td></td>
</tr>
<tr>
<td>onChanging</td>
<td>textbox</td>
<td>Event: org.zkoss.zk.ui.event.InputEvent</td>
</tr>
<tr>
<td></td>
<td>datebox</td>
<td>Denotes that user is changing the content of an input component. Notice that the component's content (at the server) won't be changed until onChange is received. Thus, you have to invoke the getValue method in the InputEvent class to retrieve the temporary value.</td>
</tr>
<tr>
<td></td>
<td>decimalbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>doublebox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>combobox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bandbox</td>
<td></td>
</tr>
<tr>
<td>onSelection</td>
<td>textbox</td>
<td>Event: org.zkoss.zk.ui.event.SelectionEvent</td>
</tr>
<tr>
<td></td>
<td>datebox</td>
<td>Denotes that user is selecting a portion of the text of an input component. You can retrieve the start and end position of the selected text by use of the getStart and getEnd methods.</td>
</tr>
<tr>
<td></td>
<td>decimalbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>doublebox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>combobox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bandbox</td>
<td></td>
</tr>
<tr>
<td>onFocus</td>
<td>textbox</td>
<td>Event: org.zkoss.zk.ui.event.Event</td>
</tr>
<tr>
<td></td>
<td>datebox</td>
<td>Denotes when a component gets the focus.</td>
</tr>
<tr>
<td></td>
<td>decimalbox</td>
<td>Remember event listeners execute at the server, so the focus at the client might be changed when the event listener for onFocus got executed.</td>
</tr>
<tr>
<td></td>
<td>doublebox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>combobox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bandbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>button</td>
<td></td>
</tr>
<tr>
<td></td>
<td>toolbarbutton</td>
<td></td>
</tr>
<tr>
<td></td>
<td>checkbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>radio</td>
<td></td>
</tr>
</tbody>
</table>
### Event Name  | Components  | Description
--- | --- | ---
**onBlur**  | textbox  
datebox  
decimalbox  
doublebox  
intbox  
combobox  
bandbox  
button  
toolbarbutton  
checkbox  
radio  | Event: org.zkoss.zk.ui.event.Event  
Denotes when a component loses the focus.  
Remember event listeners execute at the server, so the focus at the client might be changed when the event listener for **onBlur** got executed.

### List and Tree Events

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
</table>
| **onSelect** | listbox  
tabbox  
tag  
tree  | Event: org.zkoss.zk.ui.event.SelectEvent  
Denotes user has selected one or multiple child components. For listbox, it is a set of listitem. For tree, it is a set of treeitem. For tabbox, it is a tab.  
Note: **onSelect** is sent to both tag and tabbox.

| **onOpen** | groupbox  
treeitem  
combobox  
bandbox  
menupopup  
window  | Event: org.zkoss.zk.ui.event.OpenEvent  
Denotes user has opened or closed a component. Note: unlike **onClose**, this event is only a notification. The client sends this event after opening or closing the component.  
It is useful to implement *load-on-demand* by listening to the **onOpen** event, and creating components when the first time the component is opened.

### Slider and Scroll Events

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
</table>
| **onScroll** | slider  | Event: org.zkoss.zk.ui.event.ScrollEvent  
Denotes the content of a scrollable component has been scrolled by the user.
### Other Events

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onCreate</td>
<td>all</td>
<td>Denotes a component is created when rendering a ZUML page. Refer to the <strong>Component Lifecycle</strong> chapter.</td>
</tr>
<tr>
<td>onClose</td>
<td>window</td>
<td>Denotes the close button is pressed by a user, and the component shall detach itself.</td>
</tr>
<tr>
<td></td>
<td>tab</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fileupload</td>
<td></td>
</tr>
<tr>
<td>onDrop</td>
<td>all</td>
<td>Denotes another component is dropped to the component that receives this event. Refer to the <strong>Drag and Drop</strong> section.</td>
</tr>
<tr>
<td>onCheck</td>
<td>checkbox</td>
<td>Denotes the state of a component has been changed by the user.</td>
</tr>
<tr>
<td></td>
<td>radio</td>
<td></td>
</tr>
<tr>
<td></td>
<td>radiogroup</td>
<td></td>
</tr>
<tr>
<td>onMove</td>
<td>window</td>
<td>Denotes a component has been moved by the user.</td>
</tr>
<tr>
<td>onSize</td>
<td>window</td>
<td>Denotes a component has been resized by the user.</td>
</tr>
<tr>
<td>onZIndex</td>
<td>window</td>
<td>Denotes the z-index of a component has been changed by the user.</td>
</tr>
</tbody>
</table>

- **Note:** onCheck is sent to both radio and radiogroup.
<table>
<thead>
<tr>
<th>Event Name</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onTimer</td>
<td>timer</td>
<td>Event: org.zkoss.zk.ui.event.Event Denotes the timer you specified has triggered an event. To know which timer, invoke the <code>getTarget</code> method in the <code>Event</code> class.</td>
</tr>
<tr>
<td>onNotify</td>
<td>any</td>
<td>Event: org.zkoss.zk.ui.event.Event Denotes an application-dependent event. Its meaning depends on applications. Currently, no component will send this event.</td>
</tr>
<tr>
<td>onClientInfo</td>
<td>root</td>
<td>Event: org.zkoss.zk.ui.event.ClientInfoEvent Notifies a root component about the client's information, such as time zone and resolutions.</td>
</tr>
<tr>
<td>onPiggyback</td>
<td>root</td>
<td>Event: org.zkoss.zku.ui.event.Event Notifies a root component that the client has sent a request to the server. It is usually used to piggyback non-emergent UI updates to the client.</td>
</tr>
<tr>
<td>onColSize</td>
<td>columns</td>
<td>Event: org.zkoss.zul.event.ColSizeEvent Notifies the parent of a group of headers that the widths of two of its children are changed by the user.</td>
</tr>
<tr>
<td></td>
<td>listhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td>treecols</td>
<td></td>
</tr>
<tr>
<td>onPaging</td>
<td>grid</td>
<td>Event: org.zkoss.zul.event.PagingEvent Notifies one of the pages of a multi-page component is selected by the user.</td>
</tr>
<tr>
<td></td>
<td>listbox</td>
<td></td>
</tr>
<tr>
<td></td>
<td>paging</td>
<td></td>
</tr>
<tr>
<td>onUpload</td>
<td>fileupload</td>
<td>Event: org.zkoss.zul.event.UploadEvent Notifies that file(s) is uploaded, and the application can retrieve the uploaded files(s) by use of the <code>getMedia</code> or <code>getMedias</code> methods.</td>
</tr>
</tbody>
</table>

### The Event Flow of radio and radiogroup

For developer's convenience, the `onCheck` event is sent to `radio` first and then to `radiogroup`\(^{10}\). Thus, you could add listener either to the radio group or to each radio button.

---

\(^{10}\) The internal implementation is done by adding a listener when a `radio` is added to a `radiogroup`. 
The above sample has the same effect as follows.

```xml
<radiogroup onCheck="fruit.value = self.selectedItem.label">
  <radio label="Apple"/>
  <radio label="Orange"/>
</radiogroup>
You have selected : <label id="fruit"/>
```

```xml
<radiogroup>
  <radio label="Apple" onCheck="fruit.value = self.label"/>
  <radio label="Orange" onCheck="fruit.value = self.label"/>
</radiogroup>
You have selected : <label id="fruit"/>
```