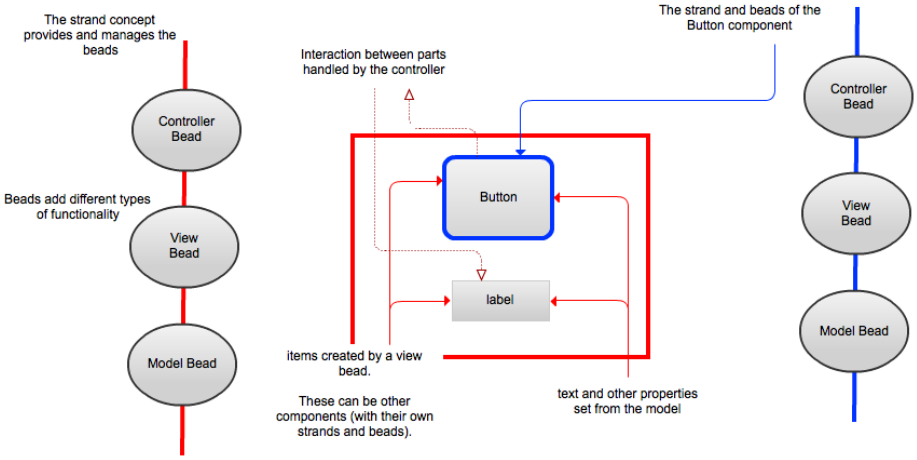
# Creating Components

## FlexJS Components

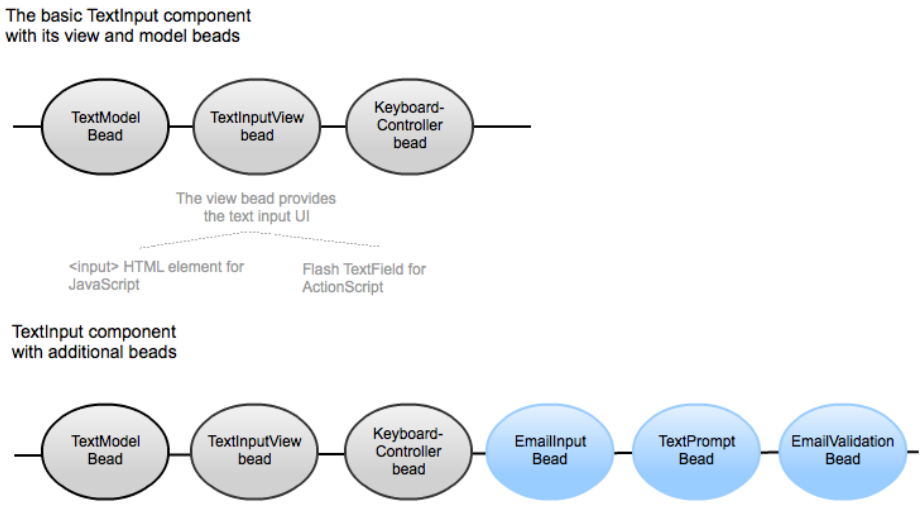
Components in the FlexJS framework are bundles of functionality formed by composition, rather than by inheritance. The concept of adding functionality (composition) is different from the current version of Flex which tends to extend (inheritance) functionality resulting in more complex components or adding features (overloading) to existing components.

For example, to have a text input field also include password and prompt features, you might use inheritance to create additional classes and wind up copying code between them. Another option is to add these properties to the text input field and overload it with features. Feature overloading makes it convenient for developers but most of the time only a basic set of features are ever used. This means applications can be very large (byte-wise) because every component has code and properties it rarely needs.




FlexJS solves these problems by allowing developers to add just the features they need when they need them. Instead of every text input field having password and prompt capabilities, only a few text input controls have those features and the application remains lighter as a result.

A component in FlexJS consists of *strands* onto which *beads* are added. A *strand* is the component wrapper while a *bead* encapsulates a particular bit of functionality. In the example above, the password and prompt features are beads that can be added to any component having a view bead that implements the `ITextFieldView` interface; the actual text input control is provided by a view bead and the data (the text itself) is managed by a model bead. In this way, components in FlexJS embody the model-view-controller (MVC) paradigm.




Beads can interact with each either through the strand, with events, or direct manipulation. For example, a bead that makes text red might listen for changes to the component's model and, when it detects a key word, changes the text in the model; the change to the model would get reflected in the view bead. Beads can add visual elements to components (*view* beads) or just change data or handle events.

## A Word About ActionScript and JavaScript

The FlexJS framework is designed to work well with ActionScript and JavaScript. The philosophy is that if it works in ActionScript, it should work in JavaScript, too, and vice-versa. You create a component (or bead) in ActionScript, designating platform-specific functionality or elements, using the COMPILE::SWF and COMPILE::JS compiler directives. See FlexJS Component Source Code Patterns for details about this development pattern for FlexJS.

You want to make things as efficient as possible in both environments. For instance, the TextInput component in ActionScript is comprised of view and model beads. The JavaScript TextInput has neither since HTML has a view and model already (via the HTMLElement); the FlexJS TextInput component is just a thin wrapper for the HTML control. As you begin creating your component, take the time to understand the best way to represent it so the component is as optimal as possible for both SWF and JS execution.

## Bead Life Cycle

There are three ways a bead can be used with a component:

1. Place an instance of the bead in-line in an MXML file using the component's <js:beads> property.
2. Reference a bead's class in a CSS style for the component.
3. Create the bead in ActionScript code (using new operator) and place it onto the component's strand using the addBead() function.

When a component is added to the display list via the addElement() function of its parent, the FlexJS framework invokes a function on the component called, addedToParent().

1. The addedToParent() function will look at the <js:beads> list and add those beads to the strand. In doing so, each bead's strand setter (see below) is called.
2. Once the <js:beads> are handled, addedToParent() then sees if there are model, view, and controller beads that need to be added to the strand. Using the model bead as an example, addedToParent() checks to see if the model is already on the strand (it may have been in the <js:beads> list). If not, then the style for the component is checked to see if iBeadModel was specified and if it is, a new model is created and added to the strand. This sequence applies to the view and controller beads as well.
3. Finally, addedToParent() dispatches the "beadsAdded" event on the component.

By first processing the <js:beads>, the FlexJS framework allows a developer to replace default beads set up in CSS. If you write your own components and have custom beads, follow this pattern:

1. Override addedToParent() and call super.addedToParent() immediately. This will perform the three steps above. It will also dispatch "beadsAdded".
2. Check to see if your custom bead is already on the strand.
3. If the bead is not on the strand, create a default bead either by looking in CSS to see if a class has been specified or just create one using the new operator. Then add it to the strand.
4. Dispatch a custom event or dispatch "initComplete" (if your component is not subclassing a container class which will dispatch "initComplete" for you) to signal that all beads are now present.

Some tips are where to perform certain tasks are given in the topics below.

## Creating a Bead

The first example shows how to make the password bead. You will see how to compose the bead and how to add it to an existing component's strand.

> This example already exists in the FlexJS library. You can look at the class files and follow along. (**See**; *flex-js/frameworks/projects/HTML/as/src/org/apache/flex/html/accessories/PasswordInputBead.as*)

For this example, the password bead will modify the underlying text control (a TextField for ActionScript and the `<input>` element for JavaScript) so that it becomes a password entry field. These are the steps needed:

1. Create the bead class (PasswordInputBead).
2. Implement the strand setter, which is required. The strand is the component (in this case, TextInput) which may or may not have all of is beads set when the strand setter is called.
3. In the strand setter, a listener is set up for the “viewChanged” event which is sent when the strand’s view bead is added. The view bead has access to the underlying text field.
4. When the “viewChanged” event is handled, the viewBead is used to get to the text field and the text field is modified to display its entry as a password.

### Create the Bead

Create a new ActionScript class called PasswordInputBead and have it implement the IBead interface. Beads do not normally have their own user interface or display; they normally use the strand. Beads can create new UI components, but then they make the strand the parent of those components.

```
package org.apache.flex.html.accessories
{
...
    public class PasswordInputBead implements IBead
```

## Implement the Strand Setter

Declare a private var called “_strand” and implement the setter function:

```
public function set strand(value:IStrand):void
{
    _strand = value;
    // more, see next paragraph
}
```

The implementation of the bead differs between JavaScript and SWF. In the browser, the underlying component is an input element and all that this bead needs to do is set its type to be "password". For the SWF platform, it is more complicated with events being intercepted and the text being changed to obscure the password. The strand setter function has these additional lines to complete it.

> The strand setter is a good place to initialize values and to set up event listeners, such as "viewChanged", "beadsAdded", or "initComplete" (or a custom event dispatched by your component).

```
COMPILE::SWF {
    IEventDispatcher(value).addEventListener("beadsAdded",beadsAddedHandler);
}
COMPILE::JS {
    var host:UIBase = value as UIBase;
    var e:HTMLInputElement = host.element as HTMLInputElement;
        e.type = 'password';
}
```

The compile directives will have the input element set up for password input with just a few lines when they are cross-compiled into JavaScript. For ActionScript, an event listener is set up to listen for when the standard beads (model, view, controller) have been added to the strand; this code is not needed in JavaScript and so it is isolated to ActionScript using the COMPILE::SWF directive.

## Implement the Event Handler

For ActionScript, create the function to handle the “beadsAddedEvent” event (note that the COMPILE::SWF directive is placed above the function definition so all of the function is included only in the ActionScript build).

```
COMPILE::SWF
private function beadsAddedHandler(event:Event):void
{
    var textView:ITextFieldView = _strand.getBeadByType(ITextFieldView) as ITextFieldView;
    if (textView) {
        var textField:CSSTextField = textView.textField;
        textField.displayAsPassword = true;
    }
}
```

The first thing the event handler does is retrieve the ITextFieldView bead (an interface, not an actual class) from the strand using the getBeadByType() function. This function will examine the strand and return the first bead that matches the given type.

With an ITextFieldView bead in hand, the next thing to do is get its underlying Flash textField and set it to display as a password. CSSTextField is a FlexJS class that extends the ActionScript TextField to make it easier to apply styles to it using CSS.

## A More Complex Bead

The PasswordInputBead is fairly simple: it just changes the type of input control using the built-in input control's property (`displayAsPassword` for ActionScript and `type = 'password'` for JavaScript). If you want the bead to do more, you can have the bead listen to events and take its own actions. Here is the `beadsAddedHandler` for a bead that restricts input to be numeric and makes sure the text being entered is valid.

```
private var _decimalSeparator:String = ".";
public function get decimalSeparator():String
{
    return _decimalSeparator;
}
public function set decimalSeparator(value:String):void
{
    if (_decimalSeparator != value) {
        _decimalSeparator = value;
    }
}
private function beadsAddedHandler(event:Event):void
{
    // get the ITextFieldView bead, which is required for this bead to work
    var textView:ITextFieldView = _strand.getBeadByType(ITextFieldView) as ITextFieldView;
    if (textView) {
        COMPILE::SWF {
            var textField:CSSTextField = textView.textField;
            textField.restrict = "0-9" + decimalSeparator;

            // listen for changes to this textField and prevent non-numeric values, such
            // as 34.09.94
            textField.addEventListener(TextEvent.TEXT_INPUT, handleTextInput);
        }
        COMPILE::JS {
            var host:UIBase = _strand as UIBase;
            var e:HTMLInputElement = host.element as HTMLInputElement;
            e.addEventListener('keypress', validateInput);
        }
    }
}
```

Like the PasswordInputBead, the ActionScript platform uses a special property of the TextField to restrict the input to numerals and a decimal separator (a property on this bead). While the `restrict` property of the TextField does limit input to those characters, it does not prevent those characters from repeating. That is, you can enter "123.45.67.890" which, of course, is not a valid number.

To have the bead act as an input validator, the bead listens for the `TEXT_INPUT` event on the TextField using the `handleTextInput()` function:

```
COMPILE::SWF
private function handleTextInput(event:TextEvent):void
{
    var insert:String = event.text;
    var caretIndex:int = (event.target as CSSTextField).caretIndex;
    var current:String = (event.target as CSSTextField).text;
    var value:String = current.substring(0,caretIndex) + insert + current.substr(caretIndex);
    var n:Number = Number(value);
    if (isNaN(n)) event.preventDefault();
}
```

Since the `TEXT_INPUT` event is dispatched before the text property of the TextField is changed, you can determine if the combination of the new text in `event.text` plus the TextField's current text value, is a valid number. After converting the combined text to a Number, it can be tested to see if the String represents a numeric value and, if not, the default action of the event is prevented: actually adding the new text to the TextField.

When the event's default action is prevented, not only is the input prevented from being added to the TextField, the model for the component is also unchanged.

The JavaScript version of this bead works similarly to the JavaScript PasswordInputBead where you set things up in the strand setter and listen for the appropriate event, a `keypress` event in this case as shown in the COMPILE::JS code block.

Unlike ActionScript, JavaScript input elements (prior to HTML 5) do not have any way to restrict input characters, so you have to do that yourself, which is done in the `validateInput()` event handler:

```
COMPILE::JS
private function validateInput(event:Event):void
{
  var code = event.charCode;

  // backspace or delete
  if (event.keyCode == 8 || event.keyCode == 46) return;

  // tab or return/enter
  if (event.keyCode == 9 || event.keyCode == 13) return;

  // left or right cursor arrow
  if (event.keyCode == 37 || event.keyCode == 39) return;

  var key = String.fromCharCode( code );

  var regex = /[0-9]|\./;
  if( !regex.test(key) ) {
    event.returnValue = false;
    if(event.preventDefault) event.preventDefault();
    return;
  }
  var cursorStart = event.target.selectionStart;
  var cursorEnd   = event.target.selectionEnd;
  var left = event.target.value.substring(0,cursorStart);
  var right = event.target.value.substr(cursorEnd);
  var complete = left + key + right;
  if (isNaN(complete)) {
    event.returnValue = false;
    if(event.preventDefault) event.preventDefault();
  }
```

The top portion of the function allows special keys to go through unaltered (backspace, tab, return, delete, and arrow keys). The next test determines if the character is '0' through '9' or the period. If the character fails that test, the event is prevented (same as ActionScript). If the character is valid, it is inserted into the input string and that assemblage is tested. If the string now fails a numeric test (`isNaN()` can take either a number or a string) the event is also prevented.

Imagine if every text input control had the ability to restrict its input just by providing some options. This more complex bead is a good example of how FlexJS allows you to build an application using only the parts you need. Having only a handful of TextInput components be able to restrict or validate their data makes for a smaller application footprint. Further, by extracting the functionality of numeric validation into a bead, the validation can be added to any text input control, such as a ComboBox, whenever needed.

## Using the Bead

### Updating the Manifest and Library

If your bead is to become part of the FlexJS framework library, you will need to modify additional files. If your bead is solely for your application use, you can skip this part.

Your bead must be compiled and place into the FlexJS SWC library. To do, find the FlexJSUIClasses.as file and add your bead. For example:

```
import org.apache.flex.html.accessories.PasswordInputBead; PasswordInputBead;
```

To allow your bead to use the FlexJS namespace, it must be present in the FlexJS manifest file. Find the basic-manifest.xml file and add the bead, such as:

```
<component id="PasswordInputBead" class="org.apache.flex.html.accessories.PasswordInputBead" />
```

### MXML

To make use of the bead, go to your application or initial view MXML file and create a TextInput and add the PasswordInputBead to it:

```
<js:TextInput>
        <js:beads>
                <js:PasswordInputBead />
        </js:beads>
</js:TextInput>
```

The FlexJS framework first adds any beads that are declared in MXML. After that, FlexJS adds beads for the component that are declared in a style sheet. FlexJS uses defaults.css as its style sheet where TextInputView is declared as the view bead for TextInput (more on this later). When the TextInputView bead is added, it triggers the “viewChanged” event which allows the PasswordInputBead to make its modifications.

To see how powerful adding beads can be, add another bead to this strand:

```
<js:TextInput>
        <js:beads>
                <js:PasswordInputBead />
                <based:TextPromptBead prompt=”password” />
        </js:beads>
</js:TextInput>
```

When the application is run, not only can you enter text as a password, a prompt will appear as long as the field is empty.

Of course, if you find that you frequently need a set of components, creating a composite, custom component that combines the beads needed is also possible; add the beads in your custom component's addedToParent() override function.

## Bead Guidelines

- Most beads use their strand for their UI parent. Beads can make or use other components or visual parts, then they add them to their strand’s display list.
- Try to use interfaces whenever possible. This keeps implementation code separate and lets modules work better.
- Use the paradigm of separation of concerns as much as possible. That is, do not have beads do more than necessary and separate functions into multiple beads. Very often a component will have one bead for the visual display (a view bead), one bead to hold its data (a model bead), and one bead to handle user interactions (a control bead). This allows a developer to swap out beads as necessary. For example, a mobile developer might want to replace the Slider's mouse controller bead with a touch controller bead: the component's other parts (track, thumb, model, etc.) remain the same.

## Making a Component

In the previous example, a new bead was created and added to an existing bead. These steps show you how to make a new component. Remember that components are usually made up of a model bead, a view bead, and a control bead, but this depends on what the component does.

The Slider component is a good example of the MVC architecture of FlexJS components. The Slider has a view bead (which manages the track and thumb - also beads), a model bead (the `RangeModel` used by other components), and control bead (to handle mouse clicks and drags). Once you've decided what your component will do, break it down into those three parts. In general, your view and controller beads will probably be unique to your component but you might be able to re-use a model bead.

Using Slider as an example, the basic approach to building a FlexJS component is:

- Slider.as (and Slider.js) is the *strand* and provides the UI display list. The strand part extends UIBase (for ActionScript) or creates the base element (for JavaScript) and defines the public properties and events for the component. In the case of Slider, it has property setters and getters for `minimum`, `maximum`, `value`, and `snapInterval` that pass through to the model. Slider also dispatches a `valueChanged` event.
- SliderView.as is the view bead and provides the track and thumb, which are also beads; SliderView simply manages them.
- SliderMouseController is the controller bead and watches for mouse events on the SliderView, translating them to model values and updating the view.
  The separation of the parts makes it easy for someone to replace them. For example, you could change how the Slider looks or behaves by replacing the view or controller beads.
- RangeModel.as (and RangeModel.js) provide the data model for the component. The model has property setters and getters for `minimum`, `maximum`, `value`, and `snapInterval` which are called by the strand (Slider.as and Slider.js). Changes to any of the properties dispatch a corresponding change event.

One thing you will not see in the Slider.as (strand) code is the application or naming of any specific beads. The beads for a component are identified by the style for the bead. If you look at the `defaults.css` file, you'll find the Slider has the following style defined:

```
Slider
{
    IBeadModel: ClassReference("org.apache.flex.html.staticControls.beads.models.RangeModel");
    iBeadView:  ClassReference("org.apache.flex.html.staticControls.beads.SliderView");
    iBeadController: ClassReference("org.apache.flex.html.staticControls.beads.controllers.
SliderMouseController");
    iThumbView: ClassReference("org.apache.flex.html.staticControls.beads.SliderThumbView");
    iTrackView: ClassReference("org.apache.flex.html.staticControls.beads.SliderTrackView");
}
```

A very powerful piece of FlexJS is the `ValuesManager` which connects components to styles and applies the beads. In this style definition for Slider, you can see which model, view, and controller are being applied. To replace something, just change it in the style. Since the SliderView uses beads for the thumb and track, those are also defined in the style so you can replace those elements, too.

When you make your own components, be sure to think about replacement - what parts of your component do you think someone might want to change or swap out - then place them into a style definition for your component.

## Slider (strand)

If you open Slider.as, you'll see that it implements the setters and getters, but all they do is call upon the model's corresponding setters and getters. Your strand really needs to provide the public interface and that's about it. The strand should not create any views or manipulate any values. There are probably going to be exceptions, but this should be the rule: keep functionality as separate as possible and join it together with events.

The Slide extends UIBase, which on the SWF side will be DisplayObject and on the JavaScript side will be a  element. That is fine for Slider, but in case you want to use a different HTML element, you can override the createElement() function which is present only for the JavaScript side:

```
/**
 * @flexjsignorecoercion org.apache.flex.core.WrappedHTMLElement
 */
COMPILE::JS
override protected function createElement():WrappedHTMLElement
{
    element = document.createElement('div') as WrappedHTMLElement;
    element.style.width = '200px';
    element.style.height = 30px';
    // set anything else here
    return element;
}
```

Notice the use of *flexjsignorecoercion* doc tag in the comment. This tells the cross-compiler not to generate JavaScript for the "as" construct involving WrappedHTMLElement.

## SliderView (view bead)

If you open the SliderView.as file and look at the strand setter function, you can see how the track and thumb beads are identified and added.

```
_track = new Button();
Button(_track).addBead(new (ValuesManager.valuesImpl.getValue(_strand, "iTrackView")) as IBead);

_thumb = new Button();
Button(_thumb).addBead(new (ValuesManager.valuesImpl.getValue(_strand, "iThumbView")) as IBead);
```

Both the track and thumb parts of the Slider are Buttons. Since Buttons are also FlexJS components and follow the same pattern, they too have beads. The look of the track and slider are encapsulated in Button-compatible beads so they are fetched from the style definition by the ValuesManager and added to the Button's strand.

Once the pieces of the view are created, event listeners are set up so the view knows what's happening to the strand, especially the size and the value. Changes to the size cause the view to layout out its children. Changes to the value position the thumb over the track.

> *Be sure to add any sub-components to the strand's display list shortly after creating them. This will fire off those sub-components' life cycles and allow them to trigger their own compositions. This way, when it comes time for your view bead to size and position the sub-elements, they will have dimension.*

## RangeModel (model bead)

If you open the RangeModel.as (or any of the other model files) you'll see they are a collection of property setters and getters (and their backing variables, of course). Every property has an event dispatched when it changes. Note that the event is dispatched from the model, not its strand. Beads that need to listen for changes to the model should fetch the strand's model and set up listeners for those properties it is interested in. For example, the Slider's mouse controller bead can update the model value which will be picked up the Slider's view bead and the thumb will change position.

This is the typical pattern for getting the strand's model and setting up the listener. Note the use of interfaces instead of concrete classes.

```
// actionscript
var model:IBeadModel = _strand.getBeadByType(IBeadModel) as IBeadModel;
IEventDispatcher(model).addEventListener("valueChanged",modelListener);
```

## SliderMouseController (control bead)

If you open the SliderMouseController.as file you will see that it uses the strand setter to get the model and view. The controller's job is to coordinate the actions of the user with the model and view. A controller sets up the input event handlers (e.g., mouse events, keyboard events, touch events) to do that. The SliderMouseController sets up mouse events such that when the track is clicked, the controller catches that event, derives the value using the model and the x position of the click. The controller then updates the model with a new value. Because the model dispatches an event when the value changes, the view, listening for this event, changes the position of the thumb.

When you write your own controllers you can follow the same pattern: translate events to model values and update the model. The view bead(s) should be listening for those changes.

If you open SliderMouseController.as, you see that its `strand setter` function gets the track and thumb beads from its strand, via `getBeadByType()` and adds event listeners to them. Notice that there are differences between the ActionScript and JavaScript platforms, so these are coded accordingly.

```
COMPILE::SWF {
    var sliderView:ISliderView = value.getBeadByType(ISliderView) as ISliderView;
    sliderView.thumb.addEventListener(MouseEvent.MOUSE_DOWN, thumbDownHandler);
    sliderView.track.addEventListener(MouseEvent.CLICK, trackClickHandler, false, 99999);
}
COMPILE::JS {
    track = value.getBeadByType(
        org.apache.flex.html.staticControls.beads.SliderTrackView);
    thumb = value.getBeadByType(
        org.apache.flex.html.staticControls.beads.SliderThumbView);

    goog.events.listen(track.element, goog.events.EventType.CLICK,
                    handleTrackClick, false, this);

    goog.events.listen(thumb.element, goog.events.EventType.MOUSEDOWN,
                    handleThumbDown, false, this);

}
```

The purpose of the controller is to coordinate the input from the mouse with values from the model which are then reflected in the views. Here are the click handlers for the track:

```
COMPILE::SWF
private function trackClickHandler( event:MouseEvent ) : void
{
        event.stopImmediatePropagation();

   var sliderView:ISliderView = _strand.getBeadByType(ISliderView) as ISliderView;

        var xloc:Number = event.localX;
        var p:Number = xloc/UIBase(_strand).width;
        var n:Number = p*(rangeModel.maximum - rangeModel.minimum) + rangeModel.minimum;


        rangeModel.value = n;

        IEventDispatcher(_strand).dispatchEvent(new Event("valueChange"));
}

COMPILE::JS
private function handleTrackClick(event:BrowserEvent):void
{
    var host:Slider = _strand as Slider;
    var xloc = event.clientX;
    var p = Math.min(1, xloc / parseInt(track.element.style.width, 10));
    var n = p * (host.get_maximum() - host.get_minimum()) +
          host.get_minimum();

    host.value = n;

    origin = parseInt(thumb.element.style.left, 10);
    position = parseInt(thumb.element.style.left, 10);

    calcValFromMousePosition(event, true);

    host.dispatchEvent(new org.apache.flex.events.Event('valueChanged'));
}
```

## Using JavaScript Component Libraries with FlexJS

Many web applications now use JavaScript component sets, such as jQuery, and it is possible to use those frameworks with FlexJS; it is just a matter of changing how the components are built. The FlexJS package comes with a handful of jQuery-compatible and CreateJS-compatible components to get you started.

Remember that FlexJS works in both the ActionScript and JavaScript worlds. While you do not have to make ActionScript equivalents of your JavaScript components, it is good practice to provide them just so your application can be run in both environments. You will find ActionScript versions of the sample jQuery and CreateJS FlexJS components in the corresponding ActionScript packages: `org/apache/flex/jquery` and `org/apache/flex/createjs`, respectively.

### jQuery for FlexJS

The biggest trick is to pull in the component set definition files. At this time, FlexJS does not have an easy way to do this. It is a manual process at this point:

1. Compile your Flex application using Falcon JX.
2. Open the index.html file that was generated.
3. Insert the following lines into the <head> portion of the file.

```
<link rel="stylesheet" href="http://code.jquery.com/ui/1.10.2/themes/smoothness/jquery-ui.css" />
<script src="http://code.jquery.com/jquery-1.9.1.js"></script>
<script src="http://code.jquery.com/ui/1.10.2/jquery-ui.js"></script>
```

In the FlexJS directories you will find a small set of components in the `org/apache/flex/html/jquery` folder. If you compare these to the standard components in the `html` package, you will see there are few differences, the most obvious being the inclusion of jQuery function calls to modify the components.

Open the TextButton.as file and look for the `createElement()` and addedToParent functions in the COMPILE::JS block:

```
/**
 * @flexjsignorecoercion org.apache.flex.core.WrappedHTMLElement
 */
COMPILE::JS
override protected function createElement():WrappedHTMLElement
{
        element = document.createElement('button') as WrappedHTMLElement;
        element.setAttribute('type', 'button');

        positioner = element;
        positioner.style.position = 'relative';
        element.flexjs_wrapper = this;
        return element;
}

COMPILE::JS
override public function addedToParent():void
{
        super.addedToParent();
        $(element).button();
}
```

The only difference between the jQuery version of the TextButton and the basic Text button is the jQuery button-making function, `$(element).button()`. Your jQuery components may differ greatly from their non-jQuery

Once you've created or converted the components to use jQuery, the JavaScript side of FlexJS can use jQuery functions to manipulate the components and their beads. For example, might want to create a bead that manipulates the styles and provides effects such as fades. Rather than modifying every component to have the ability to fade in and out, you can just add the "FadeBead" to a component strand.

## CreateJS for FlexJS

CreateJS is an altogether different way of making components from jQuery. CreateJS is more like ActionScript in that you build things rather manipulate them. As with jQuery, the big trick is to bring in the CreateJS package:

1. Compile your Flex application with the Falcon JX compiler.
2. Open the JavaScript file that was generated from your main Flex application (e.g., if your Flex application's main file is Main.mxml, open Main.js).
3. Insert the following lines before the `goog.provide()` statements:

    ```
    var mainjs = document.createElement('script');
    mainjs.src = './createjs/easeljs-0.6.0.min.js';
    document.head.appendChild(mainjs);
    ```

4. Be sure to put a copy of the CreateJS component files in a sub-directory called, `createjs`, next to the other source directories.

If you open the TextButton.js file in the CreateJS package, you will find two definitions for the TextButton class. One in a COMPILE::SWF block and another in a COMPILE::JS block. This was done because the two versions have enough differences that it would be awkward to mingle the compiler directive blocks; it was cleaner to define the classes separately.

In this example, CreateJS functions construct a button-like object rather than use HTML elements. While the basic, jQuery, and CreateJS TextButtons diff in implementation, they are expressed exactly the same in MXML.

## Using FlexJS jQuery and CreateJS Components in MXML

Once you've created your jQuery or CreateJS (or other component library) beads and components, you can add them to your Flex application MXML using namespaces. This is the root tag for a version of MyInitialView.mxml:

```
<js:ViewBase xmlns:fx="http://ns.adobe.com/mxml/2009"
                xmlns:basic="library://ns.apache.org/flexjs/basic"
                xmlns:jquery="library://ns.apache.org/flexjs/jquery"
                xmlns:createjs="library://ns.apache.org/flexjs/createjs"
        >
```

Once you have declared the namespaces, you use the namespaces to identify the components:

```
<!-- the basic tex button component -->
<js:TextButton label="Basic" />

<!-- the jQuery text button component -->
<jquery:TextButton label="jQuery" />

<!-- the CreateJS text button component -->
<createjs:TextButton label="CreateJS" />
```

## Cross-Compiling

The ability to generate JavaScript code and components from ActionScript sources is built into the FlexJS development patterns. You guide the compilation process through the use of COMPILE::SWF (exclude from cross-compilation, SWF only) and COMPILE::JS (exclude from SWF, cross-compile to JavaScript) directives; any code not in these blocks is cross-compiled to JavaScript and winds up in the SWF.