

Chapter 1: Text controls

Text controls in an Adobe® Flex® application can display text, let the user enter text, or do both.

About text controls

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You use Flex text-based controls to display text and to let users enter text into your application.

The following table lists the Flex text-based controls:

Control	Component set	Superclass	Multi Line	Editable	content/text/htmlText
Label	MX	UIComponent	N	N	text/htmlText
TextInput	MX	UIComponent	N	Y	text/htmlText
Text	MX	UIComponent	Y	N	text/htmlText
TextArea	MX	UIComponent	Y	Y	text/htmlText
RichTextEditor	MX	UIComponent	Y	Y	text/htmlText
TextInput	Spark	UIComponent	N	Y	text/textDisplay (textFlow)
TextArea	Spark	UIComponent	Y	Y	content/textFlow/text
Label	Spark	UIComponent	Y	N	text
RichText	Spark	UIComponent	Y	N	content/textFlow/text
RichEditableText	Spark	UIComponent	Y	Y	content/textFlow/text

How much control you have over the formatting of the text in the control depends on the type of control. The MX controls support a small set of formatting options with their style properties and the `htmlText` property. The Spark controls support a richer set of formatting options because they are based on Flash Text Engine (FTE) and Text Layout Framework (TLF).

Some MX controls can also use limited functionality of FTE and TLF. For more information, see [Using FTE in MX controls](#).

When building Flex 4 applications, you should use the Spark controls where possible. This is especially true if you plan on using TLF or embedded fonts. In some cases, there are both MX and Spark versions of the text-based control. For example, there are MX and Spark versions of the Label controls. The MX versions are in the `mx.controls` package. The Spark versions are in the `spark.components` package.

The primary Spark text controls are `Label`, `RichText`, and `RichEditableText`. The `Label` control has the least amount of functionality, but is also the most lightweight. The `RichEditableText` control has the most functionality, but also uses the most system resources. The following table shows more information about these Spark text primitives:

Feature	Spark Label	RichText	RichEditableText
Extends	UIComponent	UIComponent	UIComponent
Uses	FTE	TLF	TLF
Advanced typography	Y	Y	Y
Alpha	Y	Y	Y
Rotation	Y	Y	Y
Bi-directional text	Y	Y	Y
Default formatting with CSS styles	Y	Y	Y
Multiple lines	Y	Y	Y
Multiple formats	N	Y	Y
Multiple paragraphs	N	Y	Y
Text object model	N	Y	Y
Markup language	N	Y	Y
Inline graphics	N	Y	Y
Hyperlinks	N	N	Y
Scrolling	N	N	Y
Selection	N	N	Y
Editing	N	N	Y

Using Text Layout Framework

[Output: IPH, Print, Web] [Revision Control: Changing]

The Text Layout Framework (TLF) is a class library built on top of the Flash Text Engine (FTE). The FTE, available in Flash Player 10 and Adobe AIR 1.5, adds advanced text capabilities to Flash Player. FTE provides a set of low-level APIs for libraries that leverage these capabilities. The FTE classes are in the `flash.text.engine` package. In most cases, you will not use these classes directly.

The TLF classes are in the `flashx.textLayout` package. You are likely to use the TLF classes when you apply styles with HTML markup to text in text controls that support TLF. These controls include the RichText, Spark TextArea, and RichEditableText controls.

The primary difference between most Spark and MX text-based controls is the control over the formatting. For Spark controls, the formatting is provided by either FTE or TLF. This gives you a great deal of control over the text formatting, including support for HTML tags, columns, and bi-directional text. For MX controls, you have much more limited control over formatting by using the `htmlText` property.

The APIs of the FTE are described in detail in [Using the Flash Text Engine](#).

Features of TLF

[Output: IPH, Print, Web] [Revision Control: Changing]

Text Layout Framework (TLF) is a class library built on top of FTE. It provides high level text functionality, including:

- Bidirectional text, vertical text, and over 30 writing systems, including Arabic, Hebrew, Chinese, Japanese, Korean, Thai, Lao, and the major writing systems of India
- Selecting, editing, and flowing text across multiple columns and linked containers, as well as around inline images
- Vertical text, Tate-Chu-Yoko (horizontal within vertical text), and justifiers for East Asian typography
- Rich typographical controls, including kerning, ligatures, typographic case, digit case, digit width, and discretionary hyphens
- Cut, copy, paste, undo, and standard keyboard and mouse controls for editing
- Rich developer APIs to manipulate text content, layout, and markup and to create custom text components

When you add text to a control that uses TLF, the text is stored as a hierarchical tree of objects. The root element of this tree is the `TextFlow` class. Each node in the tree is an instance of a class defined in the `flashx.textLayout.elements` package. This tree is known as the text object model. The text object model is a subset of the FXG specification.

In the text object model, concepts such as paragraphs, spans, and hyperlinks are not represented as formats that affect the appearance of character runs in a single, central text string. Instead they are represented by runtime-accessible ActionScript objects, with their own properties, methods, and events (but not styles).

The following lists the objects you can have in a `TextFlow` object:

- Paragraphs (`ParagraphElement`)
- Images (`InlineGraphicElement`)
- Hyperlinks (`LinkElement`)
- Spans (`SpanElement`)
- TCY blocks (`TCYElement`)
- Tabs (`TabElement`) and line breaks (`BreakElement`)

If a Spark text control supports the `content` or `textFlow` properties, then you can use TLF to format and programmatically interact with the contents of that text control. If the control supports only the `text` property for its content, then you can only use a simple string for the text. The text does get parsed into the text object model. If an MX control supports the `htmlText` property, then you can use a subset of HTML tags to format the text, but its content is still a simple string that does not support the formal text object model.

You use the `content` property when you set the value of the property at compile time. This property takes a generic Object and converts it to a `TextFlow`. In general, you should avoid using the `content` property when the `textFlow` property is available.

You use the `textFlow` property when you set the value of the text at run time. This is because the content of the `textFlow` property is strongly typed as a `TextFlow` object rather than an Object.

The Spark versions of the `TextArea` and `TextInput` classes support TLF, as do the `RichText` and `RichEditableText` controls. The Spark Label control supports FTE. The default MX controls do not support TLF. However, it is possible to use TLF with some MX controls. For more information, see [Using FTE in MX controls](#).

For detailed information about using TLF and advanced typography, see [Working with text](#).

Creating TextFlow objects

[Output: IPH, Print, Web] [Revision Control: Changing]

The text object model is the tree that is created when you add content to a text-based control that supports TLF. This tree is defined by the `TextFlow` class.

When creating a `TextFlow` object, you can use the classes in the `flashx.textLayout.elements` package to provide formatting and additional functionality. These classes include `DivElement`, `ParagraphElement`, `InlineGraphicElement`, `LinkElement`, `SpanElement`, `TabElement`, `BreakElement`, and `TCYElement`.

You can create a `TextFlow` object in the following ways:

- Using the inline `content` tag attribute
- Using child tags
- Importing content with the `TextFlowUtil` class
- Using `ActionScript`

Creating a TextFlow object inline

[Output: IPH, Print, Web] [Revision Control: Changing]

You can create a `TextFlow` object by adding content to a text-based control inline. This means assigning a value to a text control's `content` tag attribute. When you do this, Flex creates a `TextFlow` object that defines the text object model for you.

This method of creating a `TextFlow` object is limited in that you cannot embed tags (such as `<p>` and `
`) inline. It is used strictly for simple blocks of text.

The following example creates a `TextFlow` object inline in a text control by using the `content` attribute:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/CreateTextFlowInline.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText id="myRET2"
    height="100" width="200"
    content="Inline TextFlow. Use this for simple blocks of text that do not require any
TLF formatting."
  />

</s:Application>
```

Creating a TextFlow object with child tags

[Output: IPH, Print, Web] [Revision Control: Changing]

A common method of creating a `TextFlow` object is to use the text control's `<s:textFlow>` or `<s:content>` child tags. This lets you structure the contents of the text control by using the supported TLF tags such as `<p>`, `<div>` and ``.

When using the `<s:textFlow>` child tag, you must be sure to also include a `<s:TextFlow>` tag within that, so that the structure of the control's uses the following structure:

```

<s:text_control>
  <s:textFlow>
    <s:TextFlow>
      <s:content .../>
    </s:TextFlow>
  </s:textFlow>
</s:text_control>

```

The following example creates two text controls; one text control defines a `TextFlow` object with the `<s:textFlow>` child tag, and the other with a `<s:content>` child tag.

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/CreateTextFlowChildTags.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText id="richTxt1" textAlign="justify" percentWidth="100">
    <s:textFlow>
      <s:TextFlow>
        <s:p fontSize="24">TextFlow Child Tag</s:p>
        <s:p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</s:p>
        <s:p>2) Cras posuere posuere sem, eu congue orci mattis quis.</s:p>
        <s:p>3) Curabitur pulvinar tellus venenatis ipsum tempus lobortis.</s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

  <s:RichEditableText id="richTxt2" textAlign="justify" percentWidth="100">
    <s:content>
      <s:p fontSize="24">Content Child Tag</s:p>
      <s:p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</s:p>
      <s:p>2) Cras posuere posuere sem, eu congue orci mattis quis.</s:p>
      <s:p>3) Curabitur pulvinar tellus venenatis ipsum tempus lobortis.</s:p>
    </s:content>
  </s:RichEditableText>

</s:Application>

```

Using the `<s:textFlow>` child tag is generally more efficient than using the `<s:content>` child tag. The contents of a text control that uses the `<s:textFlow>` child tag are strongly typed as a `TextFlow` at run time.

It is also convenient to drop the prefix on the markup tags by making the Spark library namespace the default namespace on the `<s:RichText>` tag, as the following example shows:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextFlowNamespace.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText id="richTxt1" textAlign="justify" percentWidth="100"
xmlns="library://ns.adobe.com/flex/spark">
    <textFlow>
      <TextFlow>
        <p fontSize="24">TextFlow Child Tag</p>
        <p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
        <p>2) Cras posuere posuere sem, eu congue orci mattis quis.</p>
        <p>3) Curabitur pulvinar tellus venenatis ipsum tempus lobortis.</p>
      </TextFlow>
    </textFlow>
  </s:RichEditableText>

</s:Application>

```

Alternatively, you can rely on the default MXML property of a tag to create a TextFlow object. The default MXML property of the TLF text controls is `content`. The parser converts the value of the `content` property to a TextFlow. As a result, if you do not specify a child tag, but add content inline, Flex defaults to adding text with the `content` property. The following example does not specify a child tag, but adds the contents inline:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/CreateTextFlowDefaultProperty.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText id="richTxt1" textAlign="justify" percentWidth="100">
    <s:p fontSize="24">Default MXML Property</s:p>
    <s:p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</s:p>
    <s:p>2) Cras posuere posuere sem, eu congue orci mattis quis.</s:p>
    <s:p>3) Curabitur pulvinar tellus venenatis ipsum tempus lobortis.</s:p>
  </s:RichEditableText>

</s:Application>

```

Creating a TextFlow object with the TextFlowUtil class

[Output: IPH, Print, Web] [Revision Control: Changing]

The `TextFlowUtil` class has two methods that let you add XML and strings as the content of a `TextFlow` object:

- `importFromString()`
- `importFromXML()`

Both of these methods return a `TextFlow` object. You commonly use them to add content to a TLF-based text control.

By using the `TextFlowUtil` class, you can add a locally defined `String` or `XML` object as a `TextFlow`.

The following example uses the `importFromString()` method to load a `String` object into the `TextFlow`:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextFlowMarkup.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="doSomething()" >
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script><![CDATA[
    import flashx.textLayout.elements.TextFlow;
    import spark.utils.TextFlowUtil;

    private function doSomething():void {
      var markup:String = "<p>This is paragraph 1.</p><p>This is paragraph 2.</p>";
      var flow:TextFlow = TextFlowUtil.importFromString(markup);
      myST.textFlow = flow;
    }
  ]]></fx:Script>

  <s:RichText id="myST" width="175"/>

</s:Application>
```

You can use XML as the source of a `TextFlow` in a similar manner, except that you use the `importFromXML()` method to load the XML, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextFlowXML.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="doSomething()" >
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <fx:XML id="myXML">
      <div>
        <p>This is <span fontWeight='bold'>paragraph 1</span>.</p>
        <p>This is <span fontWeight='bold'>paragraph 2</span>.</p>
      </div>
    </fx:XML>
  </fx:Declarations>

  <fx:Script><![CDATA[
    import flashx.textLayout.elements.TextFlow;
    import spark.utils.TextFlowUtil;

    private function doSomething():void {
      var flow:TextFlow = TextFlowUtil.importFromXML(myXML);
      myST.textFlow = flow;
    }
  ]]></fx:Script>

  <s:RichText id="myST" width="175"/>

</s:Application>
```

Any XML that uses valid TLF markup can be passed into a TextFlow. This includes content returned with an HTTPService call or XML loaded from an external file. The parser converts the TLF tags to TLF classes (such as SpanElement or DivElement) in the new TextFlow object.

If the first tag in an imported XML object is not a `<s:TextFlow>` tag, then the parser inserts one for you. If the first tag is a `<s:TextFlow>` tag, then you must also define the `"http://ns.adobe.com/textLayout/2008"` namespace in that tag.

The following example uses an external file that defines a TextFlow object for a text control:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0" encoding="utf-8"?>
<!-- textcontrols/ExternalXMLFile.mxml -->
<s:Application name="Spark_RichText_text_test"
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo">

  <fx:Script>
    <![CDATA[
      import spark.utils.TextFlowUtil;

      XML.ignoreWhitespace = false;
    ]]>
  </fx:Script>

  <fx:Declarations>
    <fx:XML id="textFlowAsXML" source="externalTextFlow1.xml" />
  </fx:Declarations>

  <s:RichText id="richText"
    textFlow="{TextFlowUtil.importFromXML(textFlowAsXML)}"
    horizontalCenter="0" verticalCenter="0" />

</s:Application>
```

The previous example uses the following file:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0" encoding="utf-8"?>
<!-- http://blog.flexexamples.com/2009/08/11/setting-text-in-a-spark-richtext-control-in-
flex-4/ -->
<TextFlow xmlns="http://ns.adobe.com/textLayout/2008" whiteSpaceCollapse="preserve">
  <p><span>The quick brown </span> <span fontWeight="bold">fox jumps over</span> <span> the
lazy dogg.</span></p>
</TextFlow>
```

This example used with permission from <http://blog.flexexamples.com>.

Creating a TextFlow object with the TextConverter class

[Output: IPH, Print, Web] [Revision Control: Changing]

You can use the TextConverter class to import content into a TextFlow. This is a little more advanced than using the TextFlowUtil methods to create a TextFlow object. Those methods act as wrappers around the TextConverter class.

When you import text with the TextConverter class's importToFlow() method, you can specify the format of the text. This determines how the content is stored.

The following are the types of formats you can specify when using the importToFlow() method:

- HTML
- Plain text
- TLF

If you specify TLF format, then the imported text is used to generate a text object model. If you specify plain text, then the text is not interpreted, but instead stored as a simple text string. If you specify HTML format, then the HTML tags in the text are mapped to TLF classes (for example, <p> is mapped to a ParagraphElement object in the text object model).

The following example imports a String into the TextFlow with the `importToFlow()` method:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0" encoding="utf-8"?>
<!-- textcontrols/ImportToFlowExample.mxml -->
<s:Application name="Spark_RichText_text_test"
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo">

  <fx:Script>
    <![CDATA[
      import flashx.textLayout.conversion.TextConverter;

      XML.ignoreWhitespace = false;
    ]]>
  </fx:Script>

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Declarations>
    <!-- Define a String to use with HTML and plain text format. -->
    <fx:String id="htmlTextAsHTML"><![CDATA[<p>Hello <b>world!</b></p>]]></fx:String>

    <!-- Define an XML object to use with TLF format. -->
    <fx:XML id="tfTextAsTextFlow">
      <TextFlow xmlns="http://ns.adobe.com/textLayout/2008">
        <p>Hello <span fontWeight="bold">world!</span></p>
      </TextFlow>
    </fx:XML>
  </fx:Declarations>

  <s:RichText id="richText"
    textFlow="{TextConverter.importToFlow(htmlTextAsHTML,
TextConverter.TEXT_FIELD_HTML_FORMAT)}"
    horizontalCenter="0" verticalCenter="0" />

  <s:RichText id="richText2"
    textFlow="{TextConverter.importToFlow(htmlTextAsHTML,
TextConverter.PLAIN_TEXT_FORMAT)}"
    horizontalCenter="0" verticalCenter="0" />

  <s:RichText id="richText3"
    textFlow="{TextConverter.importToFlow(tfTextAsTextFlow,
TextConverter.TEXT_LAYOUT_FORMAT)}"
    horizontalCenter="0" verticalCenter="0" />

</s:Application>
```

When you use the `TEXT_LAYOUT_FORMAT` type for the `TextConverter`, the imported text must observe the following rules:

- 1 The first tag must be a `<s:TextFlow>` tag. The parser will not insert one for you as it does with the `TextFlowUtil.importFromXML()` or `TextFlowUtil.importFromString()` methods.
- 2 The first tag must specify a namespace for the imported text. The namespace is `"http://ns.adobe.com/textLayout/2008"`.
- 3 Subsequent tags must be supported by the `TextFlow` class. For a list of supported tags, see “[HTML tags supported in TextFlow objects](#)” on page 15.

The `TextConverter` class also lets you export a `TextFlow` object’s content in a variety of formats. For an example of a text exporter, see <http://blog.flexexamples.com/2009/07/25/exporting-a-textflow-object-in-flex-4/>.

Creating a TextFlow object in ActionScript

[Output: IPH, Print, Web] [Revision Control: Changing]

You can create `TextFlow` objects that are used by TLF-based text controls in ActionScript.

If you create a `TextFlow` object in ActionScript, the `TextFlow` object requires that either the `ParagraphElement` or `DivElement` be at the top level. The following example creates two `ParagraphElement` objects and wraps them in `SpanElement` objects. It then adds them as children of a `TextFlow` object, and adds them to a `RichEditableText` control’s content:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SimpleTextModelExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="initApp()">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    import flashx.textLayout.elements.*;
    private var textFlow:TextFlow = new TextFlow();
    private var paragraph1:ParagraphElement = new ParagraphElement();
    private var paragraph2:ParagraphElement = new ParagraphElement();
    private var span1:SpanElement = new SpanElement();
    private var span2:SpanElement = new SpanElement();

    private function initApp():void {
      span1.text = "This is paragraph one in myRET.";
      span2.text = "This is paragraph two in myRET.";
      paragraph1.addChild(span1);
      paragraph2.addChild(span2);
      textFlow.addChild(paragraph1);
      textFlow.addChild(paragraph2);

      myRET.content = textFlow;
    }
  </fx:Script>

  <s:RichEditableText id="myRET" height="100" width="200">
  </s:RichEditableText>
</s:Application>
```

You can access the objects in a `TextFlow` and set the values of properties such as `color` or `direction` on them in ActionScript. The text in the controls update accordingly, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/ManipulateTextModelExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="initApp()">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    import flashx.textLayout.elements.*;
    private var textFlow:TextFlow = new TextFlow();
    private var paragraph1:ParagraphElement = new ParagraphElement();
    private var paragraph2:ParagraphElement = new ParagraphElement();
    private var span1:SpanElement = new SpanElement();
    private var span2:SpanElement = new SpanElement();

    private function initApp():void {
      span1.text = "This is paragraph one.";
      span2.text = "This is paragraph two.";
      paragraph1.addChild(span1);
      paragraph2.addChild(span2);
      textFlow.addChild(paragraph1);
      textFlow.addChild(paragraph2);

      myRET.content = textFlow;
    }

    private function changeColors():void {
      // Change color of first paragraph.
      paragraph1.color = 0xFF00FF;

      // Change color of second paragraph.
      paragraph2.setStyle("color", 0x00FF00);
    }
  </fx:Script>

  <s:RichEditableText id="myRET" height="100" width="200">
  </s:RichEditableText>

  <s:Button label="Change Colors" click="changeColors()"/>
</s:Application>
```

You can also access the objects in the text model by using the `TextFlow` class's `getElementById()` method. This requires that you set the `id` property of the flow elements, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/AccessTextFlowMethods.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="initApp()">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    import flashx.textLayout.elements.*;
    private var textFlow:TextFlow = new TextFlow();
    private var paragraph1:ParagraphElement = new ParagraphElement();
    private var paragraph2:ParagraphElement = new ParagraphElement();
    private var span1:SpanElement = new SpanElement();
    private var span2:SpanElement = new SpanElement();

    private function initApp():void {
      span1.id = "span1";
      span2.id = "span2";
      paragraph1.id = "paragraph1";
      paragraph2.id = "paragraph2";

      span1.text = "This is paragraph one.";
      span2.text = "This is paragraph two.";
      paragraph1.addChild(span1);
      paragraph2.addChild(span2);
      textFlow.addChild(paragraph1);
      textFlow.addChild(paragraph2);

      myRET.content = textFlow;
    }

    private function changeColors():void {
      // Set color of paragraph one.
      textFlow.getElementByID("paragraph1").setStyle("color", 0xFF0000);
      // Set color of paragraph two.
      textFlow.getElementByID("paragraph2").setStyle("color", 0xFF0000);
    }
  </fx:Script>

  <s:RichEditableText id="myRET" height="100" width="200">
</s:RichEditableText>

  <s:Button label="Change Colors" click="changeColors()" />
</s:Application>
```

In the previous example, because the text flow elements are created programmatically, their `id` properties are also set explicitly on the objects that represent the span and paragraph elements. If you were creating the text flow with HTML markup, you would specify the value of the `id` property in the HTML tag (for example, ``).

HTML tags supported in TextFlow objects

[Output: IPH, Print, Web] [Revision Control: Changing]

The text object model converts HTML elements to TLF classes. For example, if you specify a <p> tag in your text control's content, then the parser converts it to a ParagraphElement in the control's TextFlow.

The following table describes the supported elements of the TLF text object model:

Element	Class	Description
div	DivElement	A division of text; can contain only div or p elements.
p	ParagraphElement	A paragraph; can contain any element except div.
a	LinkElement	A hypertext link, also known as an anchor; can contain the tcy, span, img, tab, and br elements.
tcy	TCYElement	A run of horizontal text, used in vertical text such as Japanese; contain the a, span, img, tab, or br elements.
span	SpanElement	A run of text in a paragraph; cannot contain other elements.
img	InlineGraphicElement	An image in a paragraph element.
tab	TabElement	A tab character.
br	BreakElement	A break character; text continues on the next line, but does not start a new paragraph.

Adding images with TLF

[Output: IPH, Print, Web] [Revision Control: Changing]

TLF supports embedding images in text controls by using the InlineGraphicElement class.

The InlineGraphicElement class can point to an image file, such as a GIF, JPG, or PNG file.

You specify the location of the image by using the source property. The location can be relative to the deployed location of the application (for example, "images/butterfly.gif") or it can be a full path to the image (for example, "http://yourserver.com/images/butterfly.gif").

The following example loads a simple image from a local location:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SimpleInlineGraphic.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="doSomething()" >

  <fx:Script>
    <![CDATA[
      import flashx.textLayout.elements.*;
      import flashx.textLayout.*;

      [Bindable]
      private var textFlow:TextFlow;

      private var img:InlineGraphicElement;

      private function doSomething():void {
        textFlow = new TextFlow();

        var p2:ParagraphElement = new ParagraphElement();
        img = new InlineGraphicElement();
        img.source = "../assets/butterfly.gif";
        img.height = 100;
        img.width = 100;
        p2.addChild(img);
        textFlow.addChild(p2);
      }
    ]]>
  </fx:Script>

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:Panel title="Simple Inline Graphic Image"
    width="90%" height="90%"
    horizontalCenter="0" verticalCenter="0">

    <s:RichEditableText id="richTxt" textAlign="justify" width="100%"
      textFlow="{textFlow}" />

  </s:Panel>

</s:Application>
```

As with text-based examples, you can use a variety of techniques to load the image with an `InlineGraphicElement` object. The following example uses child tags:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SimpleInlineGraphicTags.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:Panel title="Simple Inline Graphic Image"
    width="90%" height="90%"
    horizontalCenter="0" verticalCenter="0">

    <s:RichEditableText id="richTxt" textAlign="justify" width="100%">
      <s:textFlow>
        <s:TextFlow>
          <s:p>
            <s:img source="../assets/butterfly.gif" height="100" width="100"/>
          </s:p>
        </s:TextFlow>
      </s:textFlow>
    </s:RichEditableText>
  </s:Panel>

</s:Application>
```

Rather than load the image at run time, you can also use an embedded image for the `InlineGraphicImage` object's source, as the following example shows:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SimpleEmbed.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    [Embed(source="../assets/butterfly.gif")]
    [Bindable]
    public var imgCls:Class;
  </fx:Script>

  <s:RichEditableText id="richText" textAlign="justify" width="100%">
    <s:textFlow>
      <s:TextFlow>
        <s:p>
          <s:img id="myImage" source="{imgCls}" height="100" width="100"/>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

</s:Application>

```

The `InlineGraphicElement` class can also display a drawn element, such as a `Sprite` or an `FXG` component. The following example loads an `FXG` component into the `InlineGraphicElement`:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/FXGInlineGraphic.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:comps="comps.*"
  creationComplete="doSomething()">

  <fx:Script>
    <![CDATA[
      import flashx.textLayout.elements.*;
      import flashx.textLayout.*;
      import comps.*;

      [Bindable]
      private var textFlow:TextFlow;

      private var img:InlineGraphicElement;

      private function doSomething():void {
        textFlow = new TextFlow();

```

```

        var p2:ParagraphElement = new ParagraphElement();
        img = new InlineGraphicElement();
        img.source = new ArrowAbsolute();
        img.height = 100;
        img.width = 100;
        p2.addChild(img);
        textFlow.addChild(p2);
    }
    ]]>
</fx:Script>

<s:layout>
    <s:VerticalLayout/>
</s:layout>

<s:Panel title="Simple Inline Graphic Image"
width="90%" height="90%"
horizontalCenter="0" verticalCenter="0">

    <s:RichEditableText id="richTxt" textAlign="justify" width="100%"
        textFlow="{textFlow}" />

</s:Panel>

</s:Application>

```

The FXG component used in this example is as follows:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0" encoding="utf-8"?>
<!-- fvg/comps/ArrowAbsolute.fvg -->
<fvg:Graphic xmlns:fvg="http://ns.adobe.com/fvg/2008" version="1">
    <!-- Use Use compact syntax with absolute coordinates. -->
    <fvg:Path data="
        M 20 0
        C 50 0 50 35 20 35
        L 15 35
        L 15 45
        L 0 32
        L 15 19
        L 15 29
        L 20 29
        C 44 29 44 6 20 6
    ">
    <!-- Define the border color of the arrow. -->
    <fvg:stroke>
        <fvg:SolidColorStroke color="0x888888"/>
    </fvg:stroke>
    <!-- Define the fill for the arrow. -->
    <fvg:fill>
        <fvg:LinearGradient rotation="90">
            <fvg:GradientEntry color="0x000000" alpha="0.8"/>
            <fvg:GradientEntry color="0xFFFFFFFF" alpha="0.8"/>
        </fvg:LinearGradient>
    </fvg:fill>
</fvg:Path>
</fvg:Graphic>

```

For more information about using FXG, see Flash XML Graphics (FXG) and MXML graphics.

You must explicitly specify the `height` or `width` properties of the `InlineGraphicElement` object (or both); otherwise, the image will not appear.

If you do not know the image's dimensions, then you can use the `InlineGraphicElement` class's `measuredHeight` and `measuredWidth` properties to define the size of the image. You can only use these properties after the image loads. To do this, you can trigger a function that sizes the image off of the `TextFlow`'s `StatusChangeEvent` event, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/LoadImageEvent.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="doSomething()">

  <fx:Script>
    <![CDATA[
      import flashx.textLayout.events.StatusChangeEvent;
      import flashx.textLayout.elements.*;
      import flashx.textLayout.*;

      [Bindable]
      private var textFlow:TextFlow;

      private var img:InlineGraphicElement;

      private function doSomething():void {
        textFlow = new TextFlow();
        textFlow.addEventListener(StatusChangeEvent.INLINE_GRAPHIC_STATUS_CHANGE,
sizeGraphic);

        var p2:ParagraphElement = new ParagraphElement();
        img = new InlineGraphicElement();
        //img.source = "http://www.adobe.com/ubi/globalnav/include/adobe-lq.png";
        img.source = "../assets/butterfly.gif";
        p2.addChild(img);
        textFlow.addChild(p2);
      }
      private function sizeGraphic(e:StatusChangeEvent):void {
        if (e.status == "ready" || e.status == "sizePending") {
          img.height = img.measuredHeight;

```

```
        img.width = img.measuredWidth;
    }
}
]]>
</fx:Script>

<s:layout>
    <s:VerticalLayout/>
</s:layout>

<s:Panel title="Sizing Inline Graphic Image"
    width="90%" height="90%"
    horizontalCenter="0" verticalCenter="0">

    <s:RichEditableText id="richTxt" textAlign="justify" width="100%"
        textFlow="{textFlow}" />

</s:Panel>

</s:Application>
```

You can get a reference to the `InlineGraphicElement`'s `graphic` by using the `graphic` property. This property points to the embedded `DisplayObject`. The following example changes the `alpha` of the `DisplayObject` when you move the slider:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/DisplayObjectImageElement.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    private function changeAlpha():void {
      myImage.graphic.alpha = .5;
    }
  </fx:Script>

  <s:RichEditableText id="richText" textAlign="justify" width="100%">
    <s:textFlow>
      <s:TextFlow>
        <s:p>
          <s:img id="myImage" source="../assets/butterfly.gif" height="100"
width="100"/>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

  <s:HSlider id="hSlider" minimum="0" maximum="1" value="1" stepSize="0.1"
snapInterval="0.1" liveDragging="true"
valueCommit="myImage.graphic.alpha=hSlider.value;"/>

</s:Application>
```

Note that the previous example uses a RichEditableText control and not a RichText control. The RichEditableText control supports user interaction, while the RichText control does not.

Adding hyperlinks with TLF

[Output: IPH, Print, Web] [Revision Control: Changing]

TLF based text controls support hyperlinks with the LinkElement class. Hyperlinks by themselves do nothing. You must add event handlers that react to user interaction.

To insert a LinkElement object in the TextFlow, you use the <a> tag.

The default behavior of a LinkElement object in the text object model is to launch a new browser window and navigate to the location specified in the href property of the LinkElement object. The following example shows a basic navigation link:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SimpleLinkElement.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText id="richTxt"
    editable="false"
    focusEnabled="false"
  >
    <s:textFlow>
      <s:TextFlow>
        <s:p>
          The following link takes you to: <s:a
href="http://www.adobe.com">Adobe.com</s:a>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

  <s:RichEditableText id="richTxt2"
    focusEnabled="false"
  >
    <s:textFlow>
      <s:TextFlow>
        <s:p>
          Hold CTRL key down when using the following link: <s:a
href="http://www.adobe.com">Adobe.com</s:a>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

</s:Application>

```

To ensure that your links use the hand cursor when the user mouses over them, set the `editable` property to `false` on the `RichTextEditor` object. If you do not, then your user must hold the Control key down while moving the cursor over the link to be able to click on that link.

If you want to prevent the default behavior of the `LinkElement` object, you can suppress the click event in the event handler. You do this by calling the `stopPropagation()` and `preventDefault()` methods in the event handler.

The `stopPropagation()` method prevents processing of any event listeners in nodes subsequent to the current node in the event flow. The `preventDefault()` method cancels the event's default behavior.

You can then access properties of the `LinkElement` object to carry out additional actions. The following example presents the user with an option of navigating to the target location or cancelling the action:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/CustomLinkElementHandling.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    import flashx.textLayout.events.FlowElementMouseEvent;
    import flashx.textLayout.elements.LinkElement;
    import mx.controls.Alert;
    import mx.events.CloseEvent;

    private var linkTarget:String;

    private function doSomething(e:FlowElementMouseEvent):void {
      e.stopImmediatePropagation();
      e.preventDefault();

      var le:LinkElement = e.flowElement as LinkElement;
      linkTarget = le.href;

      Alert.show("You are about to navigate away from this page.", "Alert", Alert.OK |
Alert.CANCEL, this, alertListener, null, Alert.OK);
    }

    private function alertListener(e:CloseEvent):void {
      if (e.detail == Alert.OK) {
        navigateToURL(new URLRequest(linkTarget), '_self')
      }
    }
  </fx:Script>

  <s:RichEditableText id="richTxt"
    editable="false"
    focusEnabled="false"
  >
    <s:textFlow>
      <s:TextFlow>
        <s:p>
          The following link takes you to: <s:a href="http://www.adobe.com"
target="_blank" click="doSomething(event)">Adobe.com</s:a>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

</s:Application>
```

The default styling for a `LinkElement` is underlined, blue text. To style the links in TLF, you can use a `SpanElement` object and set style properties on that. The following example removes the underline, and makes the link color red rather than blue:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/StyledLinkElement.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText id="richTxt"
    editable="false"
    focusEnabled="false">
    <s:textFlow>
      <s:TextFlow>
        <s:p>
          The following link takes you to: <s:a href="http://www.adobe.com"><s:span
color="0xFF0066" textDecoration="none">Adobe.com</s:span></s:a>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

</s:Application>
```

You can also use the `TextLayoutFormat` class to style hyperlinks. For more information, see “[Styling TLF-based text controls](#)” on page 25.

Styling TLF-based text controls

[Output: IPH, Print, Web] [Revision Control: Changing]

The Spark text controls `Label`, `RichText`, and `RichEditableText` support specifying default text formatting with CSS styles. The complete set of styles supports all of TLF’s formatting capabilities, including kerning and bidirectionality.

The names and descriptions of the styles supported by the TLF text controls are the described in the `flashx.textLayout.formats.TextLayoutFormat` and `flashx.textLayout.edit.SelectionFormat` classes.

The default values for these styles is defined by the `global` selector in the `defaults.css` file. This file is compiled into the `framework.swc` file. To change the values of the defaults, you can create your own global selector; for example:

```
<fx:Styles>
  global {
    fontFamily: "Verdana"
  }
</fx:Style>
```

Some styles are inheriting, meaning that if you set them on a container they affect the children of that container. Generally, the choice of whether a CSS style is inheriting or non-inheriting is made based on whether the corresponding TLF format inherits from the parent `FlowElement` to the child `FlowElement`.

Because a skinnable component is a type of Spark container, inheriting styles are inherited by children of the container. If you set an inheriting style such as `fontSize` on a Spark Button, for example, the style is applied to the Label class that renders the Button's label. The Label control is defined in the ButtonSkin class. The parts that make up a larger component are known as subcomponents. If you set a non-inheriting style such as `backgroundColor` it will not affect the Label subcomponent of a Button; instead you can create a custom type selector for Label, or you can reskin the Button and set the value of the `backgroundColor` property directly on the Label.

The following example shows an inheriting style and a non-inheriting style set on the Button control. The inheriting style, `color`, is applied to the text, but the non-inheriting style, `backgroundColor`, is not.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TLFStyles.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";

    s|Button {
      color: red;
      paddingLeft: 10;
    }
  </fx:Style>
  <s:Button label="Click Me"/>
</s:Application>
```

To set the value of a non-inheriting style property on the Button's label, you can set the value of the `backgroundColor` property in a Label type selector. The following example sets the `backgroundColor` style to `#33CC99` for the Label type, which affects the way the Button control's label is rendered because the Label class is a subcomponent of Button:

<codeblock> Auto-update of code-reference is turned off.

```
<?xml version="1.0"?>
<!-- textcontrols/TLFStylesSubComps.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";

    s|Button {
      color: red;
    }

    s|Label {
      paddingLeft: 10;
    }
  </fx:Style>
  <s:Button label="Click Me"/>
</s:Application>
```

For the `TextArea` and `TextInput` controls, the text is rendered by a `RichEditableText` subcomponent. You can access the `RichEditableText` subcomponent by using the `textDisplay` property. You can then call the `setStyle()` method on this property, which applies non-inheritable style properties to the subcomponent.

The following example illustrates the difference between correctly and incorrectly applying a non-inheriting style property to a pair of `TextArea` controls:

<codeblock> Auto-update of code-reference is turned off.

```
<?xml version="1.0"?>
<!-- textcontrols/TextAreaStyling.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="doSomething()">

  <fx:Script>
    private function doSomething():void {
      /* To set a non-inheritable style on a TextArea, you must actually
       apply it to the underlying RichEditableText subcontrol, which is
       accessed through the textDisplay property: */
      text1.textDisplay.setStyle("columnCount", 2);

      /* Setting a non-inheritable style directly on the TextArea does
       not apply the style properly. */
      text2.setStyle("columnCount", 2);
    }
  </fx:Script>

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
```

```
<s:TextArea id="text1" width="200" height="100">
  <s:textFlow>
    <s:TextFlow>
      <s:p>This is TextArea #1. This is enough text to ensure that there will be more
than one column if the columnCount property is properly applied.</s:p>
    </s:TextFlow>
  </s:textFlow>
</s:TextArea>

<s:TextArea id="text2" width="200" height="100">
  <s:textFlow>
    <s:TextFlow>
      <s:p>This is TextArea #2. This is enough text to ensure that there will be more
than one column if the columnCount property is properly applied.</s:p>
    </s:TextFlow>
  </s:textFlow>
</s:TextArea>

</s:Application>
```

TLF's FlowElement tags such as `<p>` and ``, which are supported in the content of the RichText or RichEditableText controls, support formatting only with properties, not using CSS. As a result, to set styles on individual flow elements, you must set them as you would set a style property, but not in CSS.

The following example shows setting style properties on the paragraphs. It shows you that you can set the style properties either with the `setStyle()` method or as a property assignment.

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TLFStylesProperties.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    import flashx.textLayout.elements.*;

    private function initApp():void {
      // You cannot set these properties in CSS.

      // Set color of first paragraph to red with the setStyle() method.
      myRET.textFlow.getChildAtIndex(0).setStyle("color", 0xFF0000);
      // Set color of second paragraph to green with a property assignment.
      myRET.textFlow.getChildAtIndex(1).color = 0x00FF00;
    }
  </fx:Script>

  <s:RichEditableText id="myRET" height="100" width="200">
    <s:content>
      <s:p id="p1">This is paragraph 1.</s:p>
      <s:p id="p2">This is paragraph 2.</s:p>
    </s:content>
  </s:RichEditableText>

  <s:Button click="initApp()" />
</s:Application>

```

The RichText and RichEditableText controls support additional style properties that Label does not. These properties include properties related to columns and paragraphs, including `columnCount`, `columnGap`, `paragraphSpaceAfter`, `paragraphStartIndent`, and `whiteSpaceCollapse`.

The RichEditableText control supports style properties that are not supported by the Label and RichText controls. These properties include `selectionColor`, `inactiveSelectionColor`, and `unfocusedSelectionColor`.

Applying styles with the TextLayoutFormat class

[Output: IPH, Print, Web] [Revision Control: Changing]

To style TLF-based text controls, you can also use the TextLayoutFormat class. You begin by creating an instance of the TextLayoutFormat class. You can then set any formatting properties on that object, including paragraph indentation, leading, justification, and typographic case. Finally, you specify the custom TextLayoutFormat with the TextFlow object's `hostFormat` property.

The following example sets various styles on the text:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextLayoutFormatExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="initApp()">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    import flashx.textLayout.formats.*;

    private function initApp():void {
      var textLayoutFormat:TextLayoutFormat = new TextLayoutFormat();
      textLayoutFormat.color = 0x336633;
      textLayoutFormat.fontFamily = "Arial, Helvetica, _sans";
      textLayoutFormat.fontSize = 14;
      textLayoutFormat.paragraphSpaceBefore = 15;
      textLayoutFormat.paragraphSpaceAfter = 15;
      textLayoutFormat.typographicCase = TLFTypographicCase.LOWERCASE_TO_SMALL_CAPS;

      textFlow.hostFormat = textLayoutFormat;
    }
  </fx:Script>

  <s:RichEditableText id="richTxt"
    editable="false"
    focusEnabled="false">
    <s:textFlow>
      <s:TextFlow id="textFlow">
        <s:p>
          The following link takes you to <s:a
href="http://www.adobe.com">Adobe.com</s:a>
        </s:p>
        <s:p>
          The following link takes you to <s:a
href="http://www.omniture.com">Omniture.com</s:a>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

</s:Application>

```

You can define a `TextLayoutFormat` object inline, rather than in ActionScript. The following example defines `TextLayoutFormat` objects that style the hyperlink, based on its state:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/StyledLinkElement2.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText id="richText"
    editable="false"
    focusEnabled="false">
    <s:textFlow>
      <s:TextFlow>
        <s:p>
          <s:linkHoverFormat>
            <s:TextLayoutFormat color="#33CC00" textDecoration="underline"/>
          </s:linkHoverFormat>
          <s:linkNormalFormat>
            <s:TextLayoutFormat color="#009900"/>
          </s:linkNormalFormat>
          <s:a href="http://www.adobe.com">Adobe.com</s:a>
        </s:p>
      </s:TextFlow>
    </s:textFlow>
  </s:RichEditableText>

</s:Application>

```

Applying styles with the Configuration class

[Output: IPH, Print, Web] [Revision Control: Changing]

Another approach to styling TLF is to use the Configuration class. This class provides access to some properties of text, such as its state. For example, you can use this class to define the appearance of a hyperlink when it is active or when it is hovered over. You can also use this class to define the appearance of text when it is selected.

To use a Configuration object with TLF, you define the styles on the TextLayoutFormat or SelectionFormat objects and assign them to the Configuration object's format properties.

The format properties include the following:

- defaultLinkActiveFormat
- defaultLinkHoverFormat
- defaultLinkNormalFormat
- focusedSelectionFormat
- inactiveSelectionFormat
- textFlowInitialFormat
- unfocusedSelectionFormat

The following example defines the appearance of the hyperlink by using custom TextLayoutFormat objects attached to a Configuration object:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0" encoding="utf-8"?>
<!-- textcontrols/StylingWithConfig.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               creationComplete="initApp()">
  <fx:Script>
    <![CDATA[
      import flashx.textLayout.conversion.TextConverter;
      import flashx.textLayout.elements.Configuration;
      import flashx.textLayout.elements.IConfiguration;
      import flashx.textLayout.formats.ITextLayoutFormat;
      import flashx.textLayout.formats.TextDecoration;
      import flashx.textLayout.formats.TextLayoutFormat;

      private function initApp():void {
        var txt:String = "Check out our website at <a
href='http://www.adobe.com/'>adobe.com</a>.";

        var cfg:Configuration = new Configuration(true);

        var normalFmt:TextLayoutFormat = new
TextLayoutFormat(cfg.defaultLinkNormalFormat);
        normalFmt.color = 0xFF0000; // red
        normalFmt.textDecoration = TextDecoration.NONE;

        var hoverFmt:TextLayoutFormat = new TextLayoutFormat(cfg.defaultLinkHoverFormat);
        hoverFmt.color = 0xFF00FF; // purple
        hoverFmt.textDecoration = TextDecoration.UNDERLINE;

        cfg.defaultLinkNormalFormat = normalFmt;
        cfg.defaultLinkHoverFormat = hoverFmt;

        rt.textFlow = TextConverter.importToFlow(txt,
TextConverter.TEXT_FIELD_HTML_FORMAT, cfg);
      }
    ]]>
  </fx:Script>

  <s:RichEditableText id="rt" x="20" y="20" editable="false" />
</s:Application>

```

Skinning TLF-based text controls

[Output: IPH, Print, Web] [Revision Control: Changing]

You do not typically add skins or chrome to the Spark text controls.

The Label, RichText, and RichEditableText Spark text controls are used in the skins of skinnable components. Because each has a different set of features, you can use the lightest weight text control that meets your needs.

For example, the default skin of a Spark Button uses the Label class to render the label of the Button. If you have a Button that requires rich text, you can replace the Label control in its skin with a RichText control.

The default skins of the Spark TextInput and TextArea use a RichEditableText control to provide an area in which text can be edited. The border and background are provided by a Rect class, and the scrollbars by a Scroller class.

The Spark TextArea and TextInput controls provide TLF-based functionality with additional chrome. For example, if you need a RichEditableText control with a border and focus halo, then you can use the Spark TextArea control.

For more information on skinning Spark controls, see [Creating Spark Skins](#).

Mirroring and bidirectional text

[Output: IPH, Print, Web] [Revision Control: Changing]

Some languages, such as Hebrew and Arabic, read from right to left. However, those languages often include text from other languages that read from left to right. Parts of a sentence in Hebrew, for example, might read from right to left, but they might include English words that are read from left to right. This makes it necessary for Flex controls to support bidirectional text.

To use bidirectional text in a TLF text-based control, use the `direction` style.

Bidirectional text lets you render text from left-to-right (LTR) or right-to-left (RTL). You can embed RTL text inside an LTR block so that portions of a paragraph render LTR and portions render RTL. There is a dominant direction for the entire paragraph, but parts of the paragraph can be read in the opposite direction, and this can be nested.

The following simple example embeds a small amount of Hebrew text in a text control, and sets the `direction` style to LTR. The text is written and stored using the UTF-8 encoding.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0" encoding="utf-8"?>
<!-- textcontrols/RTLHebrewExample.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo">
    <s:layout><s:VerticalLayout/></s:layout>
    <s:Graphic>
        <s:Label direction="flashx.textLayout.formats.Direction.RTL" text="???? ?????? ??????
????? ?????? This is a Hebrew test" width="100" height="100"/>
    </s:Graphic>
</s:Application>
```

The following example includes English and two examples of RTL text (Hebrew and Arabic). The text is defined as Unicode characters, which makes it easier to follow. You can click the button to toggle the RichEditableText control from RTL to LTR.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RTLTest.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="addText()">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
</s:Application>
<fx:Style>
  @namespace s "library://ns.adobe.com/flex/spark";
  @font-face {
    src:url("../assets/MyriadWebPro.ttf");
    fontFamily: myFontFamily;
    advancedAntiAliasing: true;
    cff: true;
    unicodeRange:
      U+0041-005A, /* Latin upper-case [A..Z] */
      U+0061-007A, /* Latin lower-case a-z */
      U+002E-002E, /* Latin period [.] */
      U+05E1,      /* The necessary Hebrew letters */
      U+05B5,
      U+05E4,
      U+05B6,
      U+05E8,
      U+0645,      /* The necessary Arabic letters */
      U+062F,
      U+0631,
      U+0633,
      U+0629;
  }

  s|RichText {
    fontFamily: myFontFamily;
    fontSize: 32;
  }
</fx:Style>

<fx:Script>
import flashx.textLayout.formats.*;
```

```

private function addText():void {
    myRT.content = "school is written " + String.fromCharCode(0x0645, 0x062f, 0x0631,
0x0633, 0x0629) +
        " in Arabic and " + String.fromCharCode(0x05E1, 0x05B5, 0x05E4, 0x05B6, 0x05E8) +
" in Hebrew.";
}
private function mirrorText():void {
    if (myRT.getStyle("direction")=="ltr") {
        myRT.setStyle("direction", flashx.textLayout.formats.Direction.RTL);
    } else {
        myRT.setStyle("direction", flashx.textLayout.formats.Direction.LTR);
    }
}
}
</fx:Script>
<s:Panel title="RTL and LTR with embedded font">
    <s:RichText id="myRT" width="400" height="150">
    </s:RichText>
</s:Panel>
<s:Button click="mirrorText()" label="Toggle Direction"/>
</s:Application>

```

Mirroring refers to laying the chrome of a component, or an entire application, out in one direction and then display in the opposite direction. For example, a mirrored `TextArea` would have its vertical scrollbar on the left. A mirrored `Tree` would have its disclosure triangles on the right. A mirrored `HGroup` would have its first child on the right, and a mirrored `TabBar` would have its first tab on the right. Mirroring can apply to subcomponents as well as chrome. For example, a mirrored `RadioButton` would have its label on the left side instead of the right side (the default).

Mirroring of components complement bidirectional text by having the application and components reflect the text direction. Mirroring is not currently supported.

Adding content to text controls

[Output: IPH, Print, Web] [Revision Control: Changing]

The Flex text-based controls let you set and get text by using the following properties:

text Plain text without formatting information. All text based controls support this property. For information on using the `text` property, see [“Using the text property”](#) on page 38.

content A rich set of HTML tags and TLF formatting. Most Spark text-based controls support this property. This property is set only, and is less efficient than the `textFlow` property. For information on using the `content` property, see [“Using the content property”](#) on page 36.

textFlow A rich set of HTML tags and TLF formatting. Most Spark text-based controls support this property. If you are using TLF, then you should use this property for supplying content to text controls rather than the `content` property. For information on using the `textFlow` property, see [“Creating TextFlow objects”](#) on page 3.

htmlText Rich text that represents formatting by using a subset of HTML tags, and can include bulleted text and URL links. Only MX text-based controls support this property. For information on using the `htmlText` property, see [“Using the htmlText property”](#) on page 41.

You can set formatted text by using the `htmlText`, `textFlow`, or `content` properties, and get it back as a plain text string by using the `text` property.

If you set more than one content-related property on a Spark text control in ActionScript (such as `text` and `content`), the last one set wins. This is similar to how `text` and `htmlText` work in MX components. If you set more than one property on an MXML tag, the winning setter cannot be predicted. The compiler does not guarantee the order in which MXML attributes get applied to a component.

Using the content property

[Output: IPH, Print, Web] [Revision Control: Changing]

You can use the `content` property to add text to TLF-based text controls. These controls include the `RichText`, `RichEditableText`, and `TextArea` controls. Text that is added with the `content` property is parsed by TLF and stored in a tree of objects. This tree is stored as a `TextFlow` object.

If you are using TLF with your text controls, then you should generally use the `textFlow` property to set the contents. The `content` property is set only, and it is less efficient than the `textFlow` property. If you set the contents of a text control with the `content` property, you can then get it with either the `textFlow` property (gets a `TextFlow` object) or the `text` property (gets a plain text `String` object). For more information about using the `textFlow` property, see “[Creating TextFlow objects](#)” on page 3.

Typically, you start the value of the `content` property with a `paragraph` tag. That tag then contains `` elements or breaks or anchor tags. TLF renders text set by the `content` property by putting it into a `span`, and that into a `paragraph`, and that into a `TextFlow`, and then rendering the `TextFlow`. The resulting tree uses the following structure:

```
<TextFlow>
  <p>
    <span>
      <contents ... >
```

When using the `content` property to add text to a control, you can use a subset of HTML in the content. This list matches the supported HTML tags for the `<s:textFlow>` tag. For a list of supported HTML tags, see “[HTML tags supported in TextFlow objects](#)” on page 15.

There are many ways to add text to a text control with the `content` property. You can add content in MXML by setting the value of the MXML tag’s `content` property, as the following example shows:

```
<s:RichText content="This is a paragraph.">
```

You can also add content with the `<s:content>` child tag, as the following example shows:

```
<s:RichText id="myRT" width="450">
  <s:content>
    This is text.
  </s:content>
</s:RichText>
```

Because the `content` property is the default property of TLF-based text controls such as `RichText`, you can omit the `<s:content>` tag in this case.

You can add also content as XML, as the following example shows:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/AddingContentAsXML.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      import flashx.textLayout.elements.*;
      import mx.utils.*;
      private var xml:XML = <p><span>This is a span</span></p>;

      private function addContent():void {
        myRET.content = xml;
      }
    ]]>
  </fx:Script>

  <s:RichEditableText id="myRET" height="100" width="200">
  </s:RichEditableText>

  <s:Button click="addContent()" label="Add Content"/>
</s:Application>

```

The content property is typed as Object because you can set it to a String, a FlowElement, or an Array of Strings and FlowElement objects. In the following example, the `` tag is one FlowElement, and the character data “world” is treated as if it were wrapped in a `<String>` tag:

```

<s:RichText fontSize="12" xmlns="library://ns.adobe.com/flex/spark">
  <span fontWeight="bold">Hello</span>World
</s:RichText>

```

You can also set the content to a single FlowElement:

```

<s:RichText fontSize="12" xmlns="library://ns.adobe.com/flex/spark">
  <span fontWeight="bold">Hello World</span>
</s:RichText>

```

or to a single String:

```

<s:RichText fontSize="12" fontWeight="bold">Hello World</s:RichText>

```

Because there is only one format across all of the content, however, it is more efficient to set the formatting on the RichText tag and the content with the text property, as the following example shows:

```

<s:RichText fontSize="12" fontWeight="bold" text="Hello World"/>

```

There is no getter for the content property. Instead, you access the host component’s textFlow property to get the parsed contents of the content property.

You can add special characters such as ampersands and angle brackets in the content property. To do this, use the same rules as described in “Specifying special characters in the text property” on page 38.

Using the text property

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You can use the `text` property to specify the text string that appears in a text control or to get the text in the control as a plain text String. All text-based Flex controls support a `text` property. When you set this property, any HTML tags in the text string appear in the control as literal text.

The interpretation of `\r`, `\n` and `\r\n` as paragraph separators is the only interpretation that occurs when setting the value of the `text` property.

You cannot specify text formatting when you set the `text` property, but you can format the text in the control, as the following example shows:

```
<s:RichText fontSize="12" fontWeight="bold" text="Hello World"/>
```

For Spark controls that support TLF, you can set the text and then manipulate it with the TLF API. You can access nodes of the text object model and format parts of the content.

For MX controls, you can set formatted text in user-editable text controls (`TextInput`, `TextArea`, `RichTextEditor`) by setting the text string with the `text` property and formatting a section of this text by using the `TextRange` class. If you get the text back by using the `htmlText` property, the property string includes HTML tags for the formatting. For more information on using the `TextRange` class, see “[Selecting and modifying text](#)” on page 52.

The following code line uses a `text` property to specify Label text:

```
<s:Label text="This is a simple text label"/>
```

You can also use a `<text>` child tag to specify text, as the following example shows:

```
<s:Label>
  <s:text>
    This is a simple text label.
  </s:text>
</s:Label>
```

The way you specify special characters, including quotation marks, greater than and less than signs, and apostrophes, depends on whether you use them in MXML tags or in ActionScript. It also depends on whether you specify the text directly or wrap the text in a CDATA section.

***Note:** If you specify the value of the `text` property by using a string directly in MXML, Flex collapses white space characters. If you specify the value of the `text` property in ActionScript, Flex does not collapse white space characters.*

Specifying special characters in the text property

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The following rules specify how to include special characters in the `text` property of a text control MXML tag, either in a property assignment, such as `text="the text"`, or in the body of a `<text>` subtag.

In standard text The following rules determine how you use special characters if you do not use a CDATA section:

- To use the special characters left angle bracket (`<`), right angle bracket (`>`), and ampersand (`&`), insert the XML character entity equivalents of `<`, `>`, and `&`, respectively. You can also use `"` and `'` for double-quotation marks (`"`) and single-quotation marks (`'`), and you can use numeric character references, such as `¥` for the Yen mark (`¥`). Do not use any other named character entities; Flex treats them as literal text.

- You cannot use the character that encloses the property text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence \" for any double-quotation marks in the string. If you surround the string in single-quotation marks (') use the escape sequence \' for any single-quotation marks in the string. You *can* use single-quotation marks inside a string that is surrounded in double-quotation marks, and double-quotation marks inside a string that is surrounded in single-quotation marks.
- Flex text controls ignore escape characters such as \t or \n in the `text` property. They ignore or convert to spaces, tabs and line breaks, depending on whether you are specifying a property assignment or a `<text>` subtag. To include line breaks, put the text in a CDATA section. In the Text control `text="string"` attribute specifications, you can also specify them as numeric character entities, such as `` for a Return character or `	` for a Tab character, but you cannot do this in an `<mx:text>` subtag.

The following code example uses the `text` property with standard text:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/StandardText.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  height="400">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <s:Label width="400"
    text="This string contains a less than, &lt;,
    greater than, &gt;, ampersand, &amp;, apostrophe, ', and
    quotation mark &quot;."/>
  <s:Label width="400"
    text='This string contains a less than, &lt;,
    greater than, &gt;, ampersand, &amp;, apostrophe, &apos;, and
    quotation mark, ".'/>
  <s:Label width="400">
    <s:text>
      This string contains a less than, &lt;, greater than,
      &gt;, ampersand, &amp;, apostrophe, ', and quotation mark, ".
    </s:text>
  </s:Label>
</s:Application>
```

The resulting application contains three almost identical text controls, each with the following text. The first two controls, however, convert any tabs in the text to spaces.

This string contains a less than, <, greater than, >, ampersand, &, apostrophe, ', and quotation mark, " .

In a CDATA section If you wrap the text string in the CDATA tag, the following rules apply:

- You cannot use a CDATA section in a property assignment statement in the text control opening tag; you must define the property in a `<text>` child tag.
- Text inside the CDATA section appears as it is entered, including white space characters. Use literal characters, such as " or < for special characters, and use standard return and tab characters. Character entities, such as `>`, and backslash-style escape characters, such as `\n`, appear as literal text.

The following code example follows these CDATA section rules:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextCDATA.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  width="500">

  <s:Label width="100%">
    <s:text>
      <![CDATA[
        This string contains a less than, <, greater than, >,
        ampersand, &, apostrophe, ', return,
        tab. and quotation mark, ".
      ]]>
    </s:text>
  </s:Label>
</s:Application>

```

The displayed text appears on three lines, as follows:

```

  This string contains a less than, <, greater than, >,
  ampersand, &, apostrophe, ', return,
  tab. and quotation mark, ".

```

Specifying special characters in ActionScript

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The following rules specify how to include special characters in a text control when you specify the control's `text` property value in ActionScript; for example, in an initialization function, or when assigning a string value to a variable that you use to populate the property:

- You cannot use the character that encloses the text string inside the string. If you surround the string in double-quotation marks ("), use the escape sequence `\"` for any double-quotation marks in the string. If you surround the string in single-quotation marks ('), use the escape sequence `\'` for any single-quotation marks in the string.
- Use backslash escape characters for special characters, including `\t` for the tab character, and `\n` or `\r` for a return/line feed character combination. You can use the escape character `\"` for the double-quotation mark and `\'` for the single-quotation mark.
- In standard text, but not in CDATA sections, you can use the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), by inserting the XML character entity equivalents of `<`, `>`, and `&`, respectively. You can also use `"` and `'` for double-quotation marks (") and single-quotation marks ('), and you can use numeric character references, such as `¥` for the Yen mark (¥). Do not use any other named character entities; Flex treats them as literal text.
- In CDATA sections only, do not use character entities or references, such as `<` or `¥` because Flex treats them as literal text. Instead, use the actual character, such as `<`.

The following example uses an initialization function to set the `text` property to a string that contains these characters:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/InitText.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  initialize="initText()">

  <fx:Script>
    public function initText():void {
      //The following is on one line.
      myText.text="This string contains a return, \n, tab, \t, and quotation mark, \". " +
        "This string also contains less than, &lt;;, greater than, &gt;;, " +
        "ampersand, &amp;, and apostrophe, ', characters.";
    }
  </fx:Script>
  <s:RichText width="450" id="myText" tabStops="100 200 300 400"/>
</s:Application>

```

The following example uses an `<fx:Script>` tag with a variable in a CDATA section to set the text property:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/VarText.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
      [Bindable]
      //The following is on one line.
      public var myText:String ="This string contains a return, \n, tab, \t, and quotation
mark, \". This string also contains less than, <, greater than, >, ampersand, &;, and
apostrophe, ', characters.";
    ]]>
  </fx:Script>
  <s:RichText width="450" text="{myText}" tabStops="100 200 300 400"/>
</s:Application>

```

The displayed text for each example appears on three lines. The first line ends at the return specified by the `\n` character. The remaining text wraps onto a third line because it is too long to fit on a single line. (Note: When you specify a tab character, you should be sure to create tab stops with the `tabStops` style property. Otherwise, the tab will instead be rendered as a carriage return. The `RichText`, `RichEditableText`, and `Spark TextArea` controls support the `tabStops` property, but the `Label` control does not.)

```

This string contains a return,
, tab,
, and quotation mark, ". This string also contains less than, <,
greater than, >, ampersand, &, and apostrophe, ', characters.

```

Using the `htmlText` property

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You use the `htmlText` property to set or get an HTML-formatted text string for MX controls. For most Spark controls, you can use the `content` property.

You can also use one tag that is not part of standard HTML, the `textFormat` tag. For details of supported tags and attributes, see “Using tags in HTML text” on page 45.

You can also specify text formatting by using Flex styles. You can set a base style, such as the font characteristics or the text weight, by using a style, and override the base style in sections of your text by using tags, such as the `` tag. In the following example, the `<mx:Text>` tag styles specify blue, italic, 14 point text, and the `<mx:htmlText>` tag includes HTML tags that override the color and point size.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/HTMLTags.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <mx:Text width="100%" color="blue" fontStyle="italic" fontSize="14">
    <mx:htmlText>
      <![CDATA[
        This is 14 point blue italic text.<br>
        <b><font color="#000000" size="10">This text is 10 point black, italic, and
bold.</font></b>
      ]]>
    </mx:htmlText>
  </mx:Text>
</s:Application>
```

Specifying HTML tags and text

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

To prevent the Flex compiler from generating errors when it encounters HTML tags in the text, use one of the following techniques:

- Wrap your text in a `CDATA` tag.
- Specify HTML markup by using the `<`, `>`, and `&` character entities in place of the left angle bracket (`<`), right angle bracket (`>`), and ampersand (`&`) HTML delimiters.

Adobe recommends using `CDATA` sections for all but simple HTML markup, because the character entity technique has significant limitations:

- Extensive HTML markup can be cumbersome to write and difficult to read.
- You must use a complex escape sequence to include the less than and ampersand characters in your text.

For example, to display the following string:

A less than character `<` and **bold text**.

without using a `CDATA` section, you must use the following text:

A less than character `<c#060;` and `bold text`.

In a `CDATA` section, you use the following text:

A less than character `<` and `bold text`.

Specifying HTML text

When you specify HTML text for a text control, the following rules apply:

- You cannot use a CDATA section directly in an inline `htmlText` property in an `<mx:Text>` tag. You must put the text in an `<mx:htmlText>` subtag, or in ActionScript code.
- Flex collapses consecutive white space characters, including return, space, and tab characters, in text that you specify in MXML property assignments or ActionScript outside of a CDATA section.
- If you specify the text in a CDATA section, you can use the text control's `condenseWhite` property to control whether Flex collapses white space. By default, the `condenseWhite` property is `false`, and Flex does not collapse white space.
- Use HTML `<p>` and `
` tags for breaks and paragraphs. In ActionScript CDATA sections you can also use `\n` escape characters.
- If your HTML text string is surrounded by single- or double-quotation marks because it is in an assignment statement (in other words, if it is not in an `<mx:htmlText>` tag), you must escape any uses of that quotation character in the string:
 - If you use double-quotation marks for the assignment delimiters, use `"` for the double-quotation mark (") character in your HTML. In ActionScript, you can also use the escape sequence `\"`.
Note: You do not need to escape double-quotation marks if you're loading text from an external file; it is only necessary if you're assigning a string of text in ActionScript.
 - If you use single-quotation marks for the assignment delimiters, use `'` for the single-quotation mark character (') in your HTML. In ActionScript, you can also use the escape sequence `\'`.
- When you enter HTML-formatted text, you must include attributes of HTML tags in double- or single-quotation marks. Attribute values without quotation marks can produce unexpected results, such as improper rendering of text. You must follow the escaping rules for quotation marks within quotation marks, as described in .

The following example shows some simple HTML formatted text, using MXML and ActionScript to specify the text:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/HTMLFormattedText.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  width="500">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script><![CDATA[
    //The following is on one line.
    [Bindable]
    public var myHtmlText:String="This string contains <b>less than </b>, &lt;;, <b>greater
than</b>, &gt;;, <b>ampersand</b>, &amp;;, and <b>double quotation mark</b>, &quot;;,
characters.";
  ]]></fx:Script>
  <mx:Text id="htmltext2" width="450" htmlText="{myHtmlText}" />
  <mx:Text width="450">
    <mx:htmlText>
      <!-- The following is on one line. Line breaks would appear in the output. -->
      <![CDATA[
        This string contains <b>less than</b>, &lt;;, <b>greater than </b>, &gt;;,
<b>ampersand</b>, &amp;;, and <b>double quotation mark</b>, &quot;;, characters.
      ]]>
    </mx:htmlText>
  </mx:Text>
</s:Application>

```

Each Text control displays the following text:

This string contains less than, <, greater than, >, ampersand, &, and double quotation mark, " characters.

Escaping special characters in HTML text

The rules for escaping special characters in HTML text differ between CDATA sections and standard text.

In CDATA sections When you specify the `htmlText` string, the following rules apply:

- In ActionScript, but not in an `<mx:htmlText>` tag, you can use standard backslash escape sequences for special characters, such as `\t` for tab and `\n` for a newline character. You can also use the backslash character to escape many special characters, such as `\\xd5` and `\"` for single- and double-quotation marks. You cannot use the combination `\<`, and a backslash before a return character has no effect on displayed text; it allows you to break the assignment statement across multiple text lines.
- In both ActionScript and the `<mx:htmlText>` tag, you can use HTML tags and numeric character entities; for example in place of `\n`, you can use a `
` tag.
- To include a left angle bracket (`<`), right angle bracket (`>`), or ampersand (`&`) character in displayed text, use the corresponding character entities: `<`, `>`, and `&`, respectively. You can also use the `"` and `'` entities for single- and double-quotation marks. These are the only named character entities that Adobe® Flash® Player and Adobe® AIR™ recognize. They recognize numeric entities, such as `¥` for the Yen mark (¥); however, they do not recognize the corresponding character entity, `¥`.

The following code example uses the `htmlText` property to display formatted text:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/HTMLTags2.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  width="500">
  <mx:Text width="100%">
    <mx:htmlText><![CDATA[<p>This string contains a <b>less than</b>, &lt;.
      </p><p>This text is in a new paragraph.<br>This is a new line.</p]]>
    </mx:htmlText>
  </mx:Text>
</s:Application>
```

This code displays the following text:

This string contains a less than, <.

This text is in a new paragraph.

This is a new line.

In standard text The following rules apply:

- You must use character entities, as described in “[Using the htmlText property](#)” on page 41, to use the left angle bracket (<), right angle bracket (>), or ampersand (&) character in HTML; for example, when you open a tag or start a character entity.
- You must use the `&` named entity combined with an HTML numeric character entity to display the less than character (use `&#060;`) and ampersand character (use `&#038;`). You can use the standard character entities, `>`, `"`, and `'`, for the greater than, double-quotation mark and single-quotation mark characters, respectively. For all other character entities, use numeric entity combinations, such as `&#165;`, for the Yen mark (¥).
- In ActionScript, but not in an `<mx:htmlText>` tag or inline `htmlText` property, you can use a backslash character to escape special characters, including the tab, newline, and quotation mark characters (but not the ampersand). In all cases, you can use (properly escaped) HTML tags and numeric character entities; for example in place of `\n`, you can use a `
` tag or `&#013;` entity.

Using tags in HTML text

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

When you use the `htmlText` property, you use a subset of HTML that is supported by Flash Player and AIR, which support the following tags:

- Anchor tag (`<a>`)
- Bold tag (``)
- Break tag (`
`)
- Font tag (``)
- Image tag (``)
- Italic tag (`<i>`)
- List item tag (``)
- Paragraph tag (`<p>`)

- Text format tag (<textformat>)
- Underline tag (<u>)

The following sections describe these tags.

Anchor tag (<a>)

The anchor <a> tag creates a hyperlink and supports the following attributes:

href Specifies the URL of the page to load in the browser. The URL can be absolute or relative to the location of the SWF file that is loading the page.

target Specifies the name of the target window to load the page into.

For example, the following HTML snippet creates the link “Go Home” to the Adobe Web site.

```
<a href='http://www.adobe.com' target='_blank'>Go Home</a>
```

The <a> tag does *not* make the link text blue or underline the text. You must apply formatting tags to change the text format. You can do this with the tag and the <u> tag.

You can also define a:link, a:hover, and a:active styles for anchor tags by using the StyleSheet class, if the component supports the styleSheet property. This property is defined on TextArea and TextField controls. The following example shows how to use it on text inside a TextArea control:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/StyleSheetExample.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            import flash.text.StyleSheet;
            private function initApp():void {
                var ss:StyleSheet = new StyleSheet;
                // Define an object for the "hover" state of the "a" tag.
                var hoverStyles:Object = new Object;
                hoverStyles.textDecoration = "underline";
                hoverStyles.color = "#FF00CC";
                // Define an object for the non-hover state of the "a" tag.
                var linkStyles:Object = new Object;
                linkStyles.color = "#FF00CC";
                // Apply the newly defined styles.
                ss.setStyle("a:hover", hoverStyles);
                ss.setStyle("a", linkStyles);
            }
        ]]>
    </fx:Script>
</s:Application>
```

```

        // Apply the StyleSheet to the TextArea control.
        myTA.styleSheet = ss;
    }
]]>
</fx:Script>

<fx:Style>
    @namespace mx "library://ns.adobe.com/flex/halo";
    mx|TextArea {
        fontFamily:Courier;
        linkcolor:#CC3300;
    }
</fx:Style>

<mx:TextArea id="myTA" height="100" width="200">
    <mx:htmlText>
        <![CDATA[<a href="http://www.adobe.com">This</a> is a link.]]>
    </mx:htmlText>
</mx:TextArea>
</s:Application>

```

MX text controls such as [Label](#), [Text](#), and [TextArea](#) can dispatch a `link` event when the user selects a hyperlink in the `htmlText` property. To generate the `link` event, prefix the hyperlink destination with `event:`, as the following example shows:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/LabelControlLinkEvent.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import flash.events.TextEvent;
            public function linkHandler(event:TextEvent):void {
                myTA.text="The link was clicked.";

                // Open the link in a new browser window.
                navigateToURL(new URLRequest(event.text), '_blank')
            }
        ]]>
    </fx:Script>

    <mx:Label selectable="true" link="linkHandler(event);">
        <mx:htmlText>
            <![CDATA[<a href='event:http://www.adobe.com'>Navigate to Adobe.com.</a>]]>
        </mx:htmlText>
    </mx:Label>
    <s:TextArea id="myTA"/>

</s:Application>

```

The Label control must have the `selectable` property set to `true` to generate the `link` event. The link event is only generated when the user clicks on the text that is wrapped in the anchor tag.

When you use the `link` event, the event is generated and the text following `event :` in the hyperlink destination is included in the `text` property of the event object. However, the hyperlink is not automatically executed; you must execute the hyperlink from within your event handler. This allows you to modify the hyperlink, or even prohibit it from occurring, in your application.

Bold tag ()

The bold `` tag renders text as bold. If you use embedded fonts, a boldface font must be available for the font or no text appears. If you use fonts that you expect to reside on the local system of your users, their system may approximate a boldface font if none exists, or it may substitute the normal font face instead of boldface. In either case, the text inside the bold tags will appear.

The following snippet applies boldface to the word *bold*:

```
This word is <b>bold</b>.
```

You cannot use the `` end tag to override bold formatting that you set for all text in a control by using the `fontWeight` style.

**Break tag (
)**

The break `
` tag creates a line break in the text. This tag has no effect in Label or TextInput controls.

The following snippet starts a new line after the word *line*:

```
The next sentence is on a new line.<br>Hello there.
```

Font tag ()

The `` tag specifies the following font characteristics: color, face, and size.

The font tag supports the following attributes:

color Specifies the text color. You must use hexadecimal (`#FFFFFF`) color values. Other formats are not supported.

face Specifies the name of the font to use. You can also specify a list of comma-separated font names, in which case Flash Player and AIR choose the first available font. If the specified font is not installed on the playback system, or isn't embedded in the SWF file, Flash Player and AIR choose a substitute font. The following example shows how to set the font face.

size Specifies the size of the font in points. You can also use relative sizes (for example, `+2` or `-4`).

The following example shows the use of the `` tag:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/FontTag.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <mx:TextArea height="100" width="250">
    <mx:htmlText>
      <![CDATA[
        You can vary the <font size='20'>font size</font>,<br><font
color="#0000FF">color</font>,<br><font face="CourierNew, Courier, Typewriter">face</font>,
or<br><font size="18" color="#FF00FF"face="Times, Times New Roman, _serif">any combination of
the three.</font>
      ]]>
    </mx:htmlText>
  </mx:TextArea>
</s:Application>
```

Image tag ()

Note: The tag is not fully supported, and might not work in some cases.

The image tag lets you embed external JPEG, GIF, PNG, and SWF files inside text fields. Text automatically flows around images you embed in text fields. This tag is supported only in dynamic and input text fields that are multiline and wrap their text.

By default, Flash displays media embedded in a text field at full size. To specify dimensions for embedded media, use the tag's `height` and `width` attributes.

In general, an image embedded in a text field appears on the line following the tag. However, when the tag is the first character in the text field, the image appears on the first line of the text field.

The tag has one required attribute, `src`, which specifies the path to an image file. All other attributes are optional.

The tag supports the following attributes:

src Specifies the URL to a GIF, JPEG, PNG, or SWF file. This attribute is required; all other attributes are optional. External files are not displayed until they have downloaded completely.

align Specifies the horizontal alignment of the embedded image within the text field. Valid values are `left` and `right`. The default value is `left`.

height Specifies the height of the image, in pixels.

hspace Specifies the amount of horizontal space that surrounds the image where no text appears. The default value is 8.

id Specifies the identifier for the imported image. This is useful if you want to control the embedded content with ActionScript.

vspace Specifies the amount of vertical space that surrounds the image where no text.

width Specifies the width of the image, in pixels. The default value is 8.

The following example shows the use of the tag and how text can flow around the image:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/ImgTag.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  width="300" height="300">
  <mx:Text height="100%" width="100%">
    <mx:htmlText>
      <![CDATA[
        <p>You can include an image in your HTML text with the &lt;img> tag.</p>
        <p><img src='assets/butterfly.gif' width='30' height='30' align='left' hspace='10'
vspace='10'>
          Here is text that follows the image. I'm extending the text by lengthening this
sentence until it's long enough to show wrapping around the bottom of the image.</p>
      ]]>
    </mx:htmlText>
  </mx:Text>
</s:Application>
```

Making hyperlinks out of embedded images

To make a hyperlink out of an embedded image, enclose the `` tag in an `<a>` tag, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/ImgTagWithHyperlink.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:TextArea width="100%" height="100%">
    <mx:htmlText>
      <![CDATA[
        <a href='http://www.adobe.com'><img src='assets/butterfly.gif' /></a>
        Click the image to go to the Adobe home page.
      ]]>
    </mx:htmlText>
  </mx:TextArea>
</s:Application>
```

When the user moves the mouse pointer over an image that is enclosed by `<a>` tags, the mouse pointer does not change automatically to a hand icon, as with standard hyperlinks. To display a hand icon, specify `buttonMode="true"` for the `TextArea` (or `Text`) control. Interactivity, such as mouse clicks and key presses, do not register in SWF files that are enclosed by `<a>` tags.

Italic tag (`<i>`)

The italic `<i>` tag displays the tagged text in italic font. If you're using embedded fonts, an italic font must be available or no text appears. If you use fonts that you expect to reside on the local system of your users, their system may approximate an italic font if none exists, or it may substitute the normal font face instead of italic. In either case, the text inside the italic tags appears.

The following snippet applies italic font to the word *italic*:

The next word is in `<i>italic</i>`.

You cannot use the `</i>` end tag to override italic formatting that you set for all text in a control by using the `fontStyle` style.

List item tag (``)

The list item `` tag ensures that the text that it encloses starts on a new line with a bullet in front of it. You cannot use it for any other type of HTML list item. The ending `` tag ensures a line break (but `` generates a single line break). Unlike in HTML, you do not surround `` tags in `` tags. For example, the following Flex code generates a bulleted list with two items:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/BulletedListExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:Text>
    <mx:htmlText >
      <![CDATA[
        <p>This is a bulleted list:<li>First Item</li><li>Second Item</li></p>
      ]]>
    </mx:htmlText>
  </mx:Text>
</s:Application>
```

Note: The `` tag does not work properly with Label controls. With TextInput controls, it must be put before the first character in the text.

Paragraph tag (`<p>`)

The paragraph `<p>` tag creates a new paragraph. The opening `<p>` tag does *not* force a line break, but the closing `</p>` tag does. Unlike in HTML, the `<p>` tag does not force a double space between paragraphs; the spacing is the same as that generated by the `
` tag.

The `<p>` tag supports the following attribute:

align Specifies alignment of text in the paragraph; valid values are `left`, `right`, `center`, and `justify`.

The following snippet generates two centered paragraphs:

```
<p align="center">This is a first centered paragraph</p>
<p align="center">This is a second centered paragraph</p>
```

Text format tag (`<textformat>`)

The text format `<textformat>` tag lets you use a subset of paragraph formatting properties of the `TextFormat` class in HTML text fields, including line leading, indentation, margins, and tab stops. You can combine text format tags with the built-in HTML tags. The text format tag supports the following attributes:

blockindent Specifies the indentation, in points, from the left margin to the text in the `<textformat>` tag body.

indent Specifies the indentation, in points, from the left margin or the block indent, if any, to the first character in the `<textformat>` tag body.

leading Specifies the amount of leading (vertical space) between lines.

leftmargin Specifies the left margin of the paragraph, in points.

rightmargin Specifies the right margin of the paragraph, in points.

tabstops Specifies custom tab stops as an array of nonnegative integers.

Underline tag (<u>)

The underline <u> tag underlines the tagged text.

The following snippet underlines the word *underlined*:

```
The next word is <u>underlined</u>.
```

You cannot use the </u> end tag to override underlining that you set for all text in a control by using the `textDecoration` style.

Selecting and modifying text

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You can select and modify text in many controls.

Selecting text

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The editable text controls provide properties and methods to select text regions and get selections. You can modify the contents of the selection as described in “[Modifying text in MX controls](#)” on page 56.

Creating a selection

The following controls let you select text:

- RichEditableText
- MX Label
- MX and Spark TextInput
- MX and Spark TextArea
- RichTextEditor and all controls that have a TextArea as a subcomponent

These controls provide the `selectRange()` (Spark controls) and `setSelection()` (MX controls) methods, which select a range of text. You specify the zero-based indexes of the start character and the position immediately *after* the last character you want to select.

The following example shows how to select the first 10 characters of various text controls:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SetSelectionTest.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="initApp()">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.core.UITextField;
      private function initApp():void {
        var tf:UITextField = haloTextArea.mx_internal::getTextField();
        var tf2:UITextField = haloTextInput.mx_internal::getTextField();
        tf.alwaysShowSelection = true;
        tf2.alwaysShowSelection = true;
      }

      private function selectText():void {
        haloTextArea.setSelection(0, 10);
        sparkTextArea.selectRange(0, 10);
        haloTextInput.setSelection(0, 10);
        sparkTextInput.selectRange(0, 10);
        sparkRET.selectRange(0, 10);

        haloRTE.textArea.setSelection(0, 10);
      }
    ]]>
  </fx:Script>
  <mx:TextArea id="haloTextArea" text="Halo TextArea control."/>
  <mx:TextInput id="haloTextInput" text="Halo TextInput control."/>
  <s:TextArea id="sparkTextArea"
    selectionHighlighting="TextSelectionHighlighting.ALWAYS"
    selectable="true"
    text="Spark TextArea control."/>
  <s:TextInput id="sparkTextInput"
    selectionHighlighting="TextSelectionHighlighting.ALWAYS"
    selectable="true"
    text="Spark TextInput control."/>
  <mx:RichTextEditor id="haloRTE" text="Spark RichTextEditor control."/>
  <s:RichEditableText id="sparkRET"
    selectionHighlighting="TextSelectionHighlighting.ALWAYS"
    selectable="true"
    text="Spark RichEditableText control."/>

  <s:Button click="selectText()" label="Select Text"/>

</s:Application>

```

To select text in a RichTextEditor control, use the control's TextArea subcomponent, which you access by using the `textArea` property. For the MX TextArea and TextInput controls, you must set the UITextField class's `alwaysShowSelection` property to `true`. Otherwise, the selection will only show when the control has focus.

For Spark text controls that support selection, you use the `selectionHighlighting` property to determine when the selection is highlighted. Possible values are `ALWAYS`, `WHEN_FOCUSED`, and `WHEN_ACTIVE`.

In addition, the MX text controls provide the `selectionBeginIndex` and `selectionEndIndex` properties, which let you set or return the zero-based location in the text of the start and position immediately *after* the end of a selection.

To select all text in a Spark text control, use the `selectAll()` method, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SelectAllExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      private function selectText(e:Event):void {
        e.currentTarget.selectAll();
      }
    ]]>
  </fx:Script>
  <s:TextArea id="sparkTextArea"
    text="Spark TextArea control." focusIn="selectText(event)"/>
  <s:TextInput id="sparkTextInput" text="Spark TextInput control."
    focusIn="selectText(event)"/>
</s:Application>
```

Getting a selection

For MX controls, you get a text control's selection by getting a [TextRange](#) object with the selected text. You can then use the `TextRange` object to modify the selected text, as described in ["Modifying text in MX controls"](#) on page 56. The technique you use to get the selection depends on the control type.

For Spark controls, you get the anchor and active positions of the selected text. You then get the value of the characters between those positions.

Get the selection in an MX TextArea or TextInput control

To get a `TextRange` object with the currently selected text in an MX `TextArea` or `TextInput` control, use the `mx.controls.textClasses.TextRange` class constructor. This text is typically stored as `htmlText` in the control. The following example displays the selections of the MX `TextArea` control when you click the button:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/ShowCurrentSelection.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="initApp()">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.core.UITextField;
      import mx.controls.textClasses.TextRange;
      private function initApp():void {
        var tf:UITextField = haloTextArea.mx_internal::getTextField();
        tf.alwaysShowSelection = true;
        haloTextArea.setSelection(0, 10);
      }

      private function getTextSelection():void {
        var myRange:TextRange = new TextRange(haloTextArea, false);
        myLabel.text = "Current Selection: \" + myRange.text + "\"";
      }
    ]]>
  </fx:Script>
  <mx:TextArea id="haloTextArea" text="Halo TextArea control."/>

  <s:Button click="getTextSelection()" label="Show Current Selection"/>
  <s:Label id="myLabel"/>

</s:Application>
```

The second parameter to the MX `TextRange` constructor, `true`, tells the constructor to return a `TextRange` object with the *selected* text. If you set this parameter to `false`, then the entire contents of the target control is added to the text range.

Get the selection in a Spark control

To get the selected text in a Spark control, you use the `selectionAnchorPosition` and `selectionActivePosition` properties of the text control. These properties define the position at the start of the selection and at the end of the selection, respectively. You use the `substring()` method on the contents of the control, and pass these positions to identify the range of characters to return.

The following example displays the selections of the Spark `TextInput` control when you click the button:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SparkTraceSelectionRanges.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="initApp()">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.core.UITextField;
      import flashx.textLayout.elements.TextRange;
      private function initApp():void {
        sparkRET.selectRange(0, 10);
      }

      private function getTextSelection():void {
        var anchorPos:int = sparkRET.selectionAnchorPosition;
        var activePos:int = sparkRET.selectionActivePosition;
        myLabel.text = "Current Selection: \" + sparkRET.text.substring(anchorPos,
activePos) + "\"";
      }
    ]]>
  </fx:Script>
  <s:RichEditableText id="sparkRET" text="Spark RichEditableText control."/>

  <s:Button click="getTextSelection()" label="Show Current Selection"/>

  <s:Label id="myLabel"/>
</s:Application>

```

Get the selection in a RichTextEditor control

Use the `selection` read-only property of the RichTextEditor to get a TextRange object with the currently selected text in its TextArea subcomponent. You can use the TextRange object to modify the selected text, as described in [Modifying text in MX controls](#). For example, to get the current selection of the MyRTE RichTextEditor control, use the following line:

```
public var mySelectedTextRange:TextRange = myRTE.selection;
```

Modifying text in MX controls

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You use the TextRange class to modify the text in MX TextArea, TextInput, or RichTextEditor controls. This class lets you affect the following text characteristics:

- `text` or `htmltext` property contents
- Text color, decoration (underlining), and alignment
- Font family, size, style (italics), and weight (bold)
- URL of an HTML `<a>` link

In Spark controls you use TLF to modify the contents, formats, and individual leaves of the text object model. For more information, see “[Using Text Layout Framework](#)” on page 2.

Getting a TextRange object

To get a `TextRange` object you use the following techniques:

- Get a `TextRange` object that contains the current text selection, as described in .
- Create a `TextRange` object that contains a specific range of text.

To create a `TextRange` object with a specific range of text, use a `TextRange` constructor with the following format:

```
new TextRange(control, modifiesSelection, beginIndex, endIndex)
```

Specify the control that contains the text, whether the `TextRange` object corresponds to a selection (that is, represents and modifies selected text), and the zero-based indexes in the text of the first and last character of the range. As a general rule, do not use the `TextRange` constructor to set a selection; use the `selectRange()` method, as described in “[Selecting text](#)” on page 52. For this reason, the second parameter should always be `false` when you specify the begin and end indexes.

To get a `TextRange` object with the fifth through twenty-fifth characters of a `TextArea` control named `myTextArea`, for example, use the following line:

```
var myTARange:TextRange = new TextRange(myTextArea, false, 4, 25);
```

Changing text

After you get a `TextRange` object, use its properties to modify the text in the range. The changes you make to the `TextRange` appear in the text control.

You can get or set the text in a `TextRange` object as HTML text or as a plain text, independent of any property that you might have used to initially set the text. If you created a `TextArea` control, for example, and set its `text` property, you can use the `TextRange.htmlText` property to get and change the text. The following example shows this usage, and shows using the `TextRange` class to access a range of text and change its properties. It also shows using `String` properties and methods to get text indexes.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextRangeExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script><![CDATA[
    import mx.controls.textClasses.TextRange
    private function resetText():void {
      ta1.text = "This is a test of the emergency broadcast system. It is only a test.";
    }
    public function alterText():void {
      // Create a TextRange object starting with "the" and ending at the
      // first period. Replace it with new formatted HTML text.
      var tr1:TextRange = new TextRange(
        ta1, false, ta1.text.indexOf("the", 0), ta1.text.indexOf(".", 0)
      );
      tr1.htmlText="<i>italic HTML text</i>"
```

```

    // Create a TextRange object with the remaining text.
    // Select the text and change its formatting.
    var tr2:TextRange = new TextRange(
        ta1, true, ta1.text.indexOf("It", 0), ta1.text.length-1
    );
    tr2.color=0xFF00FF;
    tr2.fontSize=18;
    tr2.fontStyle = "italic"; // any other value turns italic off
    tr2.fontWeight = "bold"; // any other value turns bold off
    ta1.setSelection(0, 0);
}
]]></fx:Script>
<mx:TextArea id="ta1" fontSize="12" fontWeight="bold" width="100%" height="100">
    <mx:text>
        This is a test of the emergency broadcast system. It is only a test.
    </mx:text>
</mx:TextArea>
<s:HGroup>
    <mx:Button label="Alter Text" click="alterText();" />
    <mx:Button label="Reset" click="resetText();" />
</s:HGroup>
</s:Application>

```

Example: Changing selected text in a RichTextEditor control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The following example shows how you can use the `selectedText` property of the `RichTextEditor` control to get a `TextRange` when a user selects some text, and use `TextRange` properties to get and change the characteristics of the selected text. To use the example, select a range of text with your mouse. When you release the mouse button, the string “This is replacement text.”, formatted in fuchsia Courier 20-point font replaces the selection and the text area reports on the original and replacement text.

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextRangeSelectedText.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    width="600" height="500">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script><![CDATA[
        import mx.controls.textClasses.TextRange;
        //The following text must be on a single line.
        [Bindable]
        public var htmlData:String="<textformat leading='2'><p align='center'><b><font
size='20'>HTML Formatted Text</font></b></p></textformat><br><textformat leading='2'><p
align='left'><font face='_sans' size='12' color='#000000'>This paragraph contains <b>bold</b>,
<i>italic</i>, <u>underlined</u>, and <b><i><u>bold italic underlined </u></i></b>text.
</font></p></textformat><br><p><u><font face='arial' size='14' color='#ff0000'>This a red
underlined 14-point arial font with no alignment set.</font></u></p><p align='right'><font

```

```

face='verdana' size='12' color='#006666'><b>This a teal bold 12-pt. Verdana font with alignment
set to right.</b></font></p>";
    public function changeSelectionText():void {
        //Get a TextRange with the selected text and find its length.
        var sel:TextRange = rtel.selection;
        var selLength:int = sel.endIndex - sel.beginIndex;
        //Do the following only if the user made a selection.
        if (selLength) {
            //Display the selection size and font color, size, and family.
            t1.text="Number of characters selected: " + String(selLength);
            t1.text+="\n\nOriginal Font Family: " + sel.fontFamily;
            t1.text+="\n\nOriginal Font Size: " + sel.fontSize;
            t1.text+="\n\nOriginal Font Color: " + sel.color;
            //Change font color, size, and family and replace selected text.
            sel.text="This is replacement text. "
            sel.color="fuchsia";
            sel.fontSize=20;
            sel.fontFamily="courier"
            //Show the new font color, size, and family.
            t1.text+="\n\nNew text length: " + String(sel.endIndex - sel.beginIndex);
            t1.text+="\n\nNew Font Family: " + sel.fontFamily;
            t1.text+="\n\nNew Font Size: " + sel.fontSize;
            t1.text+="\n\nNew Font Color: " + sel.color;
        }
    }
]]</fx:Script>
<!-- The text area. When you release the mouse after selecting text,
    it calls the func1 function. -->
<mx:RichTextEditor id="rtel"
    htmlText="{htmlData}"
    width="100%" height="100%"
    mouseUp="changeSelectionText()"/>

<mx:TextArea id="t1"
    editable="false"
    fontSize="12"
    fontWeight="bold"
    width="300" height="180"/>
</s:Application>

```

Label control

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The Label control is a noneditable single line of text. There are both Spark and MX versions of the Label control. The Spark version is in the spark.components package. The MX version is in the mx.controls package.

The Label control has the following characteristics:

- The user cannot change the text, but the application can modify it.
- You can specify text formatting by using HTML (MX Label only).
- You can control the alignment and sizing.
- The control's background is transparent, so the background of the component's container shows through.

- The control has no borders, so the label appears as text written directly on its background.
- The control cannot take the focus.

The differences between the MX Label and the Spark Label include the following:

- Spark Label uses FTE, while MX Label uses the TextField class for rendering.
- Spark Label offers better typography, and better support for international languages, than MX Label.
- Spark Label can display multiple lines; MX Label cannot.
- MX Label can render a limited set of HTML tags, while Spark Label can only display text with uniform formatting.
- MX Label can be made selectable, while Spark Label cannot.

For complete reference information, see the *Adobe Flex Language Reference*.

To create a multiline, noneditable text field, use a Text control. For more information, see “[Text control](#)” on page 63.

To create user-editable text fields, use the TextInput or TextArea controls. For more information, see “[TextInput control](#)” on page 60 and “[TextArea control](#)” on page 64.

Creating a Label control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You define a Label control in MXML by using the `<s:Label>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/LabelControl.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  width="150" height="80">
  <s:Label text="Label1"/>
</s:Application>
```

You use the `text` property to specify a string of plain text. To specify an HTML-formatted string, use the `htmlText` property for the MX version of the Label control. For more information on using these properties, see “[Using the text property](#)” on page 38 and “[Using the htmlText property](#)” on page 41.

Sizing a Label control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

If you do not specify a width, the Label control automatically resizes when you change the value of the `text` or `htmlText` property.

If you explicitly size a Label control so that it is not large enough to accommodate its text, the text is truncated and terminated by an ellipsis (...). The full text displays as a tooltip when you move the mouse over the Label control. If you also set a tooltip by using the `tooltip` property, the tooltip is displayed rather than the text.

TextInput control

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The `TextInput` control is a single-line text field that is optionally editable.

There are Spark and MX versions of the `TextInput` control. The Spark version is in the `spark.components` package. The MX version is in the `mx.controls` package.

The Spark version of the `TextInput` control supports TLF.

For complete reference information, see the *Adobe Flex Language Reference*.

To create a multiline, editable text field, use a `TextArea` control. For more information, see “[TextArea control](#)” on page 64. To create noneditable text fields, use the `Label` and `Text` controls. For more information, see “[Label control](#)” on page 59 and “[Text control](#)” on page 63.

The `TextInput` control does not include a label, but you can add one by using a `Label` control or by nesting the `TextInput` control in a `FormItem` container in a `Form` layout container. MX `TextInput` controls dispatch `change`, `dataChange`, `textInput`, and `enter` events, as well as the events of their parent classes. Spark `TextInput` controls dispatch all these events except the `dataChange` event.

If you disable an MX `TextInput` control, it displays its contents in a different color, represented by the `disabledColor` style. To change the disabled color for the Spark `TextInput` control, you must edit the `TextInputSkin` class. The following example shows the differences between the enabled and disabled versions of the `TextInput` controls:

<codeblock> Auto-update of code-reference is turned off.

```
<?xml version="1.0"?>
<!-- textcontrols/TextInputDisabledColors.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:VGroup>
    <mx:TextInput enabled="false" text="disabled Halo"/>
    <mx:TextInput enabled="true" text="enabled Halo"/>
    <s:TextInput enabled="false" text="disabled Spark"/>
    <s:TextInput enabled="true" text="enabled Spark"/>
  </s:VGroup>
</s:Application>
```

You can set a `TextInput` control’s `editable` property to `false` to prevent users from editing the text. You can set a `TextInput` control’s `displayAsPassword` property to conceal the input text by displaying characters as asterisks.

Creating a `TextInput` control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You define a `TextInput` control in MXML by using the `<TextInput>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an `ActionScript` block.

<codeblock> Auto-update of code-reference is turned off.

```
<?xml version="1.0"?>
<!-- textcontrols/TextInputControl.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:TextInput id="text1" width="100"/>
</s:Application>
```

You can use the `text` property to specify a string of raw text in the `TextInput` control. To specify HTML-formatted string for the Spark version, For the MX version, use the `htmlText` property. For more information, see “[Using the text property](#)” on page 38 and “[Using the htmlText property](#)” on page 41.

Sizing a TextInput control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

If you do not specify a width, the `TextInput` control automatically resizes when you change the text. It does not resize in response to typed user input.

The Spark version determines the value of its `measuredWidth` property by using a `widthInChars` property rather than measuring the text assigned to it, because the text frequently starts out empty. The value of its `measuredHeight` property is determined by the height of the font.

Binding to a TextInput control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

In some cases, you might want to bind a variable to the `text` property of a `TextInput` control so that the control represents a variable value, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/BoundTextInputControl.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script><![CDATA[
    [Bindable]
    public var myProp:String="This is the initial String myProp.";
  ]]></fx:Script>
  <s:TextInput text="{myProp}"/>
</s:Application>
```

In this example, the `TextInput` control displays the value of the `myProp` variable. Remember that you must use the `[Bindable]` metadata tag if the variable changes value and the control must track the changed values; also, the compiler generates warnings if you do not use this metadata tag. This technique works for all `textFlow`, `text`, `htmlText`, and `content` properties of the Flex text-based controls.

Using TLF with the TextInput control

[Output: IPH, Print, Web] [Revision Control: Changing]

You can use TLF with a `TextInput` control by accessing the `textDisplay` property. This property points to the underlying `RichEditableText` object that renders the text in the `TextInput` control.

The following example defines a `TextFlow` as XML, and assigns that to the `TextInput` control's `textDisplay`:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextInputTLF.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" creationComplete="doSomething()">

  <fx:Declarations>
    <fx:XML id="myXML">
      <div>
        <p>Hello <span fontWeight='bold'>world</span>!</p>
      </div>
    </fx:XML>
  </fx:Declarations>

  <fx:Script>
    import spark.utils.TextFlowUtil;

    private function doSomething():void {
      text1.textDisplay.textFlow = TextFlowUtil.importFromXML(myXML);
    }
  </fx:Script>

  <s:TextInput id="text1" width="100">
</s:TextInput>

</s:Application>
```

For more information about using TLF with Spark text controls, see “[Using Text Layout Framework](#)” on page 2.

Text control

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

Note: The Text control is an MX control. If you are using Flex 4, you should consider using the Spark RichText control.

The Text control displays multiline, noneditable text. The control has the following characteristics:

- The user cannot change the text, but the application can modify it.
- The control does not support scroll bars. If the text exceeds the control size, users can use keys to scroll the text.
- The control is transparent so that the background of the component’s container shows through.
- The control has no borders, so the label appears as text written directly on its background.
- The control supports HTML text and a variety of text and font styles.
- The text always word-wraps at the control boundaries, and is always aligned to the top of the control.

For complete reference information, see the *Adobe Flex Language Reference*.

To create a single-line, noneditable text field, use the Label control. For more information, see “[Label control](#)” on page 59. To create user-editable text fields, use the TextInput or TextArea controls. For more information, see “[TextInput control](#)” on page 60 and “[TextArea control](#)” on page 64.

Creating a Text control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You define a Text control in MXML by using the `<mx:Text>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextControl.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:Text width="175"
    text="This is an example of a multiline text string in a Text control."/>
</s:Application>
```

You use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see “Using the text property” on page 38 and “Using the htmlText property” on page 41.

This control does not support a `backgroundColor` property; its background is always the background of the control’s container.

Sizing a Text control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

Flex sizes the Text control as follows:

- If you specify a pixel value for both the `height` and `width` properties, any text that exceeds the size of the control is clipped at the border.
- If you specify an explicit pixel width, but no height, Flex wraps the text to fit the width and calculates the height to fit the required number of lines.
- If you specify a percentage-based width and no height, Flex does *not* wrap the text, and the height equals the number of lines as determined by the number of Return characters.
- If you specify only a height and no width, the height value does not affect the width calculation, and Flex sizes the control to fit the width of the maximum line.

As a general rule, if you have long text, you should specify a pixel-based `width` property. If the text might change and you want to ensure that the Text control always takes up the same space in your application, set explicit `height` and `width` properties that fit the largest expected text.

TextArea control

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The `TextArea` control is a multiline, editable text field with a border and optional scroll bars. The `TextArea` control supports the HTML and rich text rendering capabilities of Flash Player and AIR. The `TextArea` control dispatches `change` and `textInput` events.

There are Spark and MX versions of the `TextArea` control. The Spark version is in the `spark.components` package. The MX version is in the `mx.controls` package.

The Spark version of the TextArea control supports TLF.

For complete reference information, see the *Adobe Flex Language Reference*.

To create a single-line, editable text field, use the TextInput control. For more information, see “[TextInput control](#)” on page 60. To create noneditable text fields, use the Label and Text controls. For more information, see “[Label control](#)” on page 59, and “[Text control](#)” on page 63.

If you disable an MX TextArea control, it displays its contents in a different color, represented by the `disabledColor` style. To change the disabled color for the Spark TextArea control, you must edit the `TextAreaSkin` class.

You can set a TextArea control’s `editable` property to `false` to prevent editing of the text. You can set a TextArea control’s `displayAsPassword` property to conceal input text by displaying characters as asterisks.

Creating a TextArea control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You define a TextArea control in MXML by using the `<TextArea>` tag, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/TextAreaControl.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:TextArea id="textConfirm"
    width="300" height="100"
    text="Please enter your thoughts here."/>
</s:Application>
```

The MX version of the TextArea control supports the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string. For more information, see “[Using the text property](#)” on page 38 and “[Using the htmlText property](#)” on page 41.

In addition to these properties, the Spark version of the TextArea control supports the `content` property. This means that you can use TLF to format and programmatically interact with the contents of the control. For more information, see “[Using the content property](#)” on page 36.

Sizing the TextArea control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The TextArea control does not resize to fit the text that it contains.

If the new text exceeds the capacity of a TextArea control and the `horizontalScrollPolicy` property is set to `true` (the default value), the control adds a scrollbar. This applies to both the MX and Spark versions of the control.

Using TLF with the TextArea control

[Output: IPH, Print, Web] [Revision Control: Changing]

The Spark version of the TextArea control supports the advanced text formatting and manipulation of TLF. You add TLF content to the TextArea control by using either the `textFlow` or `content` properties.

The following example adds a block of TLF-formatted text to the TextArea control:

<codeblock> Auto-update of code-reference is turned off.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- textcontrols/TextAreaTLF.mxml -->
<s:Application name="RichEditableTextExample"
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo">

  <s:Panel title="TextArea TLF"
    width="90%" height="90%"
    horizontalCenter="0" verticalCenter="0">
    <s:TextArea id="ta1" textAlign="left" percentWidth="90">
      <s:textFlow>
        <s:TextFlow>
          <s:p fontSize="24">TextArea with TLF block</s:p>
          <s:p>1) Lorem ipsum dolor sit amet, consectetur adipiscing elit.</s:p>
          <s:p>2) Cras posuere posuere sem, <s:span fontWeight="bold">eu congue
orci mattis quis</s:span>.</s:p>
          <s:p>3) Curabitur <s:span textDecoration="underline">pulvinar
tellus</s:span> venenatis ipsum tempus lobortis.<s:br/>
          <s:span color="0x6600CC">Vestibulum eros velit</s:span>, bibendum
at aliquet ut.
          </s:p>
        </s:TextFlow>
      </s:textFlow>
    </s:TextArea>
  </s:Panel>
</s:Application>
```

For more information about using TLF with Spark text controls, see “[Using Text Layout Framework](#)” on page 2.

RichTextEditor control

[Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The [RichTextEditor](#) control lets users enter, edit, and format text. You apply text formatting and URL links by using subcomponents that are located at the bottom of the RichTextEditor control.

For complete reference information, see the *Adobe Flex Language Reference*.

About the RichTextEditor control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The RichTextEditor control consists of a Panel control with two direct children:

- A [TextArea](#) control in which users can enter text
- A tool bar container with format controls that let a user specify the text characteristics. Users can use the tool bar subcomponents to apply the following text characteristics:
 - Font family
 - Font size

Note: When using the `RichTextEditor` control, you specify the actual pixel value for the font size. This size is not equivalent to the relative font sizes specified in HTML by using the `size` attribute of the HTML `` tag.

- Any combination of bold, italic and underline font styles
- Text color
- Text alignment: left, center, right, or justified
- Bullets
- URL links

You use the `RichTextEditor` interactively as follows:

- Text that you type is formatted as specified by the control settings.
- To apply new formatting to existing text, select the text and set the controls to the required format.
- To create a link, select a range of text, enter the link target in the text box on the right, and press Enter. You can only specify the URL; the link always opens in a `_blank` target. Also, creating the link does not change the appearance of the link text; you must separately apply any color and underlining.
- You can cut, copy, and paste rich text within and between Flash HTML text fields, including the `RichTextEditor` control's `TextArea` subcomponent, by using the normal keyboard commands. You can copy and paste plain text between the `TextArea` and any other text application, such as your browser or a text editor.

Creating a `RichTextEditor` control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

You define a `RichTextEditor` control in MXML by using the `<mx:RichTextEditor>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RichTextEditorControl.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <mx:RichTextEditor id="myRTE" text="Congratulations, winner!" />
</s:Application>
```

You can use the `text` property to specify an unformatted text string, or the `htmlText` property to specify an HTML-formatted string. For more information on using these properties, see “[Using the text property](#)” on page 38, and “[Using the htmlText property](#)” on page 41. For information on selecting, replacing, and formatting text that is in the control, see “[Selecting and modifying text](#)” on page 52.

The following example shows the code used to create the image in “[About the RichTextEditor control](#)” on page 66:

```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RichTextEditorControlWithFormattedText.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <!-- The HTML text string used to populate the RichTextEditor control's
  TextArea subcontrol. The text is on a single line. -->
  <fx:Script><![CDATA[
    [Bindable]
    public var htmlData:String="<textformat leading='2'><p align='center'><b><font
size='20'>HTML Formatted Text</font></b></p></textformat><br><textformat leading='2'><p
align='left'><font face='_sans' size='12' color='#000000'>This paragraph contains<b>bold</b>,
<i>italic</i>, <u>underlined</u>, and <b><i><u>bold italic underlined
</u></i></b>text.</font></p></textformat><br><p><u><font face='arial' size='14'
color='#ff0000'>This a red underlined 14-point arial font with no alignment
set.</font></u></p><p align='right'><font face='verdana' size='12' color='#006666'><b>This a
teal bold 12-pt.' Verdana font with alignment set to right.</b></font></p><br><li>This is
bulleted text.</li><li><font face='arial' size='12' color='#0000ff'><u> <a
href='http://www.adobe.com'>This is a bulleted link with underline and blue color
set.</a></u></font></li>";
  ]]></fx:Script>
  <!-- The RichTextEditor control. To reference a subcontrol prefix its ID with the
RichTextEditor control ID. -->
  <mx:RichTextEditor id="rte1"
    backgroundColor="#ccffcc"
    width="500"
    headerColors="[#88bb88, #bbeebb]"
    footerColors="[#bbeebb, #88bb88]"
    title="Rich Text Editor"
    htmlText="{htmlData}"
    initialize="rte1.textArea.setStyle('backgroundColor', '0xeeffee'"
  />
</s:Application>

```

Sizing the RichTextEditor control

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

The control does not resize in response to the size of the text in the TextArea control. If the text exceeds the viewable space, by default, the TextArea control adds scroll bars. If you specify a value for either the `height` or `width` property but not both, the control uses the default value for the property that you do not set.

If you set a `width` value that results in a width less than 605 pixels wide, the RichTextEditor control stacks the subcomponents in rows.

Programming RichTextEditor subcomponents

[Chunk: No] [Output: IPH, Print, Web] [EditorialStatus: Preliminary Review]

Your application can control the settings of any of the RichTextEditor subcomponents, such as the [TextArea](#), the [ColorPicker](#), or any of the [ComboBox](#) or [Button](#) controls that control text formatting. To refer to a RichTextEditor subcomponent, prefix the requested control's ID with the RichTextEditor control ID. For example, to refer to the ColorPicker control in a RichTextEditor control that has the ID `rte1`, use `rte1.colorPicker`.

Inheritable styles that you apply directly to a RichTextEditor control affect the underlying Panel control and the subcomponents. Properties that you apply directly to a RichTextEditor control affect the underlying Panel control only.

For more information, see the [RichTextEditor](#) in the *Adobe Flex Language Reference*.

Setting RichTextEditor subcomponent properties and styles

The following simple code example shows how you can set and change the properties and styles of the RichTextEditor control and its subcomponents. This example uses styles that the RichTextEditor control inherits from the Panel class to set the colors of the Panel control header and the tool bar container, and sets the TextArea control's background color in the RichTextEditor control's creationComplete event member. When users click the buttons, their click event listeners change the TextArea control's background color and the selected color of the ColorPicker control.

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RTESubcontrol.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    height="420">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- The RichTextEditor control. To set the a subcontrol's style or property,
    fully qualify the control ID. The footerColors style sets the ControlBar colors. -->
    <mx:RichTextEditor id="rte1"
        backgroundColor="#ccffcc"
        headerColors="[#88bb88, #bbeebb]"
        footerColors="[#bbeebb, #88bb88]"
        title="Rich Text Editor"
        creationComplete="rte1.textArea.setStyle('backgroundColor','0xeeffee')"
        text="Simple sample text"
    />
    <!-- Button to set a white TextArea background. -->
    <s:Button
        label="Change appearance"
        click="rte1.textArea.setStyle('backgroundColor',
'0xffffffff');rte1.colorPicker.selectedIndex=27;"
    />
    <!-- Button to reset the display to its original appearance. -->
    <s:Button
        label="Reset Appearance"
        click="rte1.textArea.setStyle('backgroundColor',
'0xeeffee');rte1.colorPicker.selectedIndex=0;"
    />
</s:Application>
```

Removing and adding RichTextEditor subcomponents

You can remove any of the standard RichTextEditor subcomponents, such as the alignment buttons. You can also add your own subcomponents, such as a button that pops up a find-and-replace dialog box.

Remove an existing subcomponent

- 1 Create a function that calls the `removeChildAt` method of the editor's tool bar Container subcomponent, specifying the control to remove.
- 2 Call the method in the RichTextEditor control's `initialize` event listener.

The following example removes the alignment buttons from a RichTextEditor control, and shows the default appearance of a second RichTextEditor control:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RTERemoveAlignButtons.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script><![CDATA[
    public function removeAlignButtons():void {
      rt1.toolbar.removeChild(rt1.alignButtons);
    }
  ]]></fx:Script>

  <s:VGroup>
    <mx:RichTextEditor id="rt1"
      title="RichTextEditor With No Align Buttons"
      creationComplete="removeAlignButtons()"
    />
    <mx:RichTextEditor id="rt2"
      title="Default RichTextEditor"
    />
  </s:VGroup>
</s:Application>
```

Add a new subcomponent

- 1 Create an ActionScript function that defines the subcomponent. Also create any necessary methods to support the control's function.
- 2 Call the method in the RichTextEditor control's `initialize` event listener, as in the following tag:

```
<mx:RichTextEditor id="rt" initialize="addMyControl()"
```

The following example adds a find-and-replace dialog box to a RichTextEditor control. It consists of two files: the application, and a custom TitleWindow control that defines the find-and-replace dialog (which also performs the find-and-replace operation on the text). The application includes a function that adds a button to pop up the TitleWindow, as follows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/CustomRTE.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
      import mx.controls.*;
      import mx.containers.*;
      import flash.events.*;
      import mx.managers.PopUpManager;
      import mx.core.IFlexDisplayObject;
      /* The variable for the pop-up dialog box. */
      public var w:IFlexDisplayObject;
      /* Add the Find/Replace button to the Rich Text Editor control's
         toolbar container. */
      public function addFindReplaceButton():void {
        var but:Button = new mx.controls.Button();
        but.label = "Find/Replace";
        but.addEventListener("click",findReplaceDialog);
        rt.toolbar.addChild(but);
      }
      /* The event listener for the Find/Replace button's click event
         creates a pop-up with a MyTitleWindow custom control. */
      public function findReplaceDialog(event:Event):void {
        var w:MyTitleWindow = MyTitleWindow(PopUpManager.createPopUp(this, MyTitleWindow,
true));

        w.height=200;
        w.width=340;
        /* Pass the a reference to the textArea subcontrol
           so that the custom control can replace the text. */
        w.RTETextArea = rt.textArea;
        PopUpManager.centerPopUp(w);
      }
    ]]>
  </fx:Script>
  <mx:RichTextEditor id="rt" width="95%"
    title="RichTextEditor"
    text="This is a short sentence."
    initialize="addFindReplaceButton()"
  />
</s:Application>
```

The following MyTitleWindow.mxml file defines the custom myTitleWindow control that contains the find-and-replace interface and logic:

```
<codeblock> [Outputclass: NoSWF]
Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- A TitleWindow that displays the X close button. Clicking the close button
only generates a CloseEvent event, so it must handle the event to close the control. -->
<mx:TitleWindow
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark"
    title="Find/Replace"
    showCloseButton="true"
    close="closeDialog();">

    <fx:Script>
        <![CDATA[
            import mx.controls.TextArea;
            import mx.managers.PopUpManager;
            /* Reference to the RichTextArea textArea subcontrol.
             It is set by the application findReplaceDialog method
             and used in the replaceAndClose method, below. */
            public var RTETextArea:TextArea;
            /* The event handler for the Replace button's click event.
             Replace the text in the RichTextEditor TextArea and
             close the dialog box. */
            public function replaceAndClose():void{
                RTETextArea.text = RTETextArea.text.replace(ti1.text, ti2.text);
                PopUpManager.removePopUp(this);
            }
            /* The event handler for the TitleWindow close button. */
            public function closeDialog():void {
                PopUpManager.removePopUp(this);
            }
        ]]>
    </fx:Script>
    <!-- The TitleWindow subcontrols: the find and replace inputs,
         their labels, and a button to initiate the operation. -->
    <mx:Label text="Find what:"/>
    <mx:TextInput id="ti1"/>

    <mx:Label text="Replace with:"/>
    <mx:TextInput id="ti2"/>

    <mx:Button label="Replace" click="replaceAndClose();" />
</mx:TitleWindow>
```

RichText control

[Output: IPH, Print, Web] [Revision Control: Changing]

The RichText control is a middleweight Spark text control. It can display richly-formatted text, with multiple character and paragraph formats. However, it is non-interactive: it does not support scrolling, selection, or editing. If you want a control that supports formatted text plus scrolling, selection, and editing, you can use the RichEditableText control.

For specifying the text, the RichText control supports the `textFlow`, `text`, and `content` properties. If you set the `text` property, the contents are read in as a String; tags such as `<p>` and `` are ignored. If you set the `textFlow` or `content` properties, then the contents are parsed by TLF and stored as a TextFlow object.

To create a RichText control, you use the `<s:RichText>` tag in MXML, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RichTextExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <!-- You can display simple text with the text property.-->
  <s:RichText text="Hello World!"/>
  <s:RichText>
    <s:text>
      Hello World!
    </s:text>
  </s:RichText>
  <!-- You can display formatted text with the content property.-->
  <s:RichText>
    <s:content>
      Hello <s:span fontSize='16'>BIG NEW</s:span> World!
    </s:content>
  </s:RichText>
  <!-- You cannot use tags when specifying the content property inline.-->
  <s:RichText content="Hello World!"/>

</s:Application>
```

The text in a RichText control can be horizontally and vertically aligned but it cannot be scrolled. The contents of the control wraps at the right edge of the control's bounds. If the content extends below the bottom, it is clipped. It is also clipped if you turn off wrapping by setting the `lineBreak` style property to `explicit` and add content that extends past the right edge of the control.

For more information about using TLF with Spark text controls, see “[Using Text Layout Framework](#)” on page 2.

RichEditableText control

[Output: IPH, Print, Web] [Revision Control: Changing]

The RichEditableText is similar to the RichText control in that it can display richly-formatted text, with multiple character and paragraph formats. In addition, the RichEditableText control is interactive: it supports scrolling, selection, and editing.

The RichEditableText class is similar to the `spark.components.TextArea` control, except that it does not have a border, scroll bars, or focus glow.

To create a RichEditableText control in MXML, you use the RichEditableText tag, as the following example shows:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RichEditableTextExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText height="100" width="200">
    <s:text>
      Hello World!
    </s:text>
  </s:RichEditableText>
</s:Application>
```

For specifying the text, the RichEditableText control supports the `textFlow`, `text`, and `content` properties. If you set the `text` property, the contents are read in as a String; tags such as `<p>` and `` are ignored. If you set the `textFlow` or `content` properties, then the contents are parsed by TLF and stored in a TextFlow object.

The RichEditableText control also supports `insertText()` and `appendText()` methods. The content can be exported to XML using the `export()` method, which produces XML.

You can make the content of the RichEditableText control selectable by setting the `selectable` property to `true`. You can make the content of the RichEditableText control editable by setting the `editable` property to `true`.

The following example makes the RichEditableText content both selectable and editable:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/RichEditableTextExample.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:RichEditableText height="100" width="200"
    editable="true"
    selectable="true">
    <s:text>
      This text is editable and selectable!
    </s:text>
  </s:RichEditableText>
</s:Application>
```

You can read the selection range with the read-only `selectionAnchorPosition` and `selectionActivePosition` properties. You can set the selection with the `selectRange()` method. To determine the text formatting on the selection, use the `getFormatOfRange()` method. You can set the format of the selection with the `setFormatOfRange()` method.


```

<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/ScrollableRET.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:Scroller>
    <s:RichEditableText height="100" width="200"
      editable="true"
      selectable="true">
      <s:text>
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
        This text scrolls vertically!
      </s:text>
    </s:RichEditableText>
  </s:Scroller>
  <s:Label text="To enable horizontal scrolling, set lineBreak to explicit:"/>
  <s:Scroller>
    <s:RichEditableText height="100" width="200"
      editable="true"
      selectable="true"
      lineBreak="explicit">
      <s:text>
        This text scrolls horizontally! This text scrolls horizontally! This text
        scrolls horizontally!
        This text scrolls horizontally! This text scrolls horizontally! This text
        scrolls horizontally!
      </s:text>
    </s:RichEditableText>
  </s:Scroller>
</s:Application>

```

The RichEditableText control supports programmatic scrolling with its IViewport interface; it scrolls in response to the mousewheel; and it automatically scrolls as you drag-select or type more text than fits in the control's bounds. The Scroller class lets you put anything that implements the IViewport interface, such as the RichEditableText control, in it to be scrolled.

If you select a range of text that is not in the current viewport, you can programmatically scroll to that range by using the `scrollToRange()` method. The following example selects the 10th line and then scrolls to that line when you click the button:

```
<codeblock> Auto-update of code-reference is turned off.
<?xml version="1.0"?>
<!-- textcontrols/SelectScrollableRET.mxml -->
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      private function scrollToLine():void {
        myRET.selectRange(440, 471);
        myRET.scrollToRange(440, 471);
      }
    ]]>
  </fx:Script>

  <s:Scroller>
    <s:RichEditableText id="myRET"
      selectionHighlighting="TextSelectionHighlighting.ALWAYS"
      selectable="true"
      height="100" width="200">
      <s:text>
        This text scrolls vertically1!
        This text scrolls vertically2!
        This text scrolls vertically3!
        This text scrolls vertically4!
        This text scrolls vertically5!
        This text scrolls vertically6!
        This text scrolls vertically7!
        This text scrolls vertically8!
        This text scrolls vertically9!
        This text scrolls vertically10!
        This text scrolls vertically11!
        This text scrolls vertically12!
        This text scrolls vertically13!
        This text scrolls vertically14!
      </s:text>
    </s:RichEditableText>
  </s:Scroller>
  <s:Button label="Select and Scroll to Line 10" click="scrollToLine()"/>
</s:Application>
```

For more information about using TLF with Spark text controls, see [“Using Text Layout Framework”](#) on page 2.