Contents

Text Layout Framework Overview

The Challenge of Typography on the Web

Typography has a long and colorful history that stretches back beyond the advent of the printing press. Techniques for arranging letterforms on the printed page have been refined through the centuries and passed down to typographers who now practice the art with computers and laser printers. And yet, much of typography's expressive power has yet to be realized by web developers.

To better understand the challenges faced by typographers on the web, it may help to consider three factors that contribute to the difficulty of applying standard typographical techniques to the web. First, text on the web is dynamic. Readers can resize windows and change font sizes. This stands in stark contrast to the permanence of the printed page and makes it difficult to apply techniques to the web that were developed for a fixed medium. Second, the lack of strict adherence to web standards among browser vendors makes it difficult to ensure that a particular typographic design will be rendered uniformly across various browsers and platforms. Third, there is no way to ensure that a specific font is available to all browsers that may view your content.

These factors are compounded by the limited set of tools available to web designers. HyperText Markup Language (HTML) provides very limited typographical support. Although the emergence of Cascading Style Sheets (CSS) has helped to a limited extent by making it easier for web designers to assign specific fonts and type sizes to text throughout a document, CSS does not ensure that a specific font you choose will be available to all who view your web site. Moreover, each browser has an idiosyncratic implementation of both HTML and CSS, which can differ even among different versions of the same browser. To achieve a consistent look and feel across all browsers, a web designer must either employ complicated scripts to detect browsers and versions or use only the few tags and properties that are common to all browsers.

To a great extent, Flash Player offers web designers a compelling alternative to HTML and CSS. Flash Player provides better control over text attributes than does HTML and CSS. Moreover, as a browser plug-in that is not affected by the idiosyncrasies of specific browsers, Flash Player also makes it much easier to ensure a uniform look and feel across browsers and platforms. Perhaps the most significant advantage, however, is that a web designer can embed specific fonts into Flash contents.

How TextField has Pushed the Envelope

The TextField class, which first appeared as the TextField object in Flash Player 6, has been the cornerstone of Flash text handling up to and including Flash Player 9. The TextField class is relatively simple to use, and provides typographic control through the TextFormat and StyleSheet classes.

The TextField class, when used in combination with embedded fonts, was a step forward for web designers who wanted to use a specific font and to have some control over how the font is displayed. In fact, this combination served as the basis for an open source technique called Scalable Inman Flash Replacement (sIFR), which allows web designers to embed specific fonts into web content. The sIFR approach uses JavaScript to embed a SWF that replaces a small run of text within a larger body of text content. Before sIFR was invented, web designers who wanted to use a specific font for a short run of text, usually a headline, would have to create an image of the text and place that image into their web content in place of the text. The problem with text as an image is that the text size cannot change when the web page is resized and the text is not selectable, whereas the sIFR technique results in selectable text that can scale with the rest of the page.

The sIFR technique is a good example of how the TextField class was used to advance typography on the web. The technique also takes advantage of a key strength of the TextField class: Flash Player handles TextField instances that contain simple runs of static text with extreme efficiency. Longer runs of text, support for global scripts and more advanced typography, however, call for a more powerful text engine than the one used by the TextField class. Flash Player 10 provides such an engine, accessible through the new Text Layout Framework.

What the New Text Layout Framework Offers

Although TextField's ease of use and small memory footprint make it a good choice for small static text runs even in Flash Player 10, web designers eager for more advanced typography should use the new Text Layout Framework.

The Text Layout Framework is a set of ActionScript 3.0 libraries with support for complex scripts and advanced typographic and layout features not available in the TextField class.

Complex Script Support

Complex script support includes the ability to display and edit not only right-to-left scripts, but also a mixture of left-to-right and right-to-left scripts such as Arabic and Hebrew. The framework not only supports vertical text layout for Chinese, Japanese and Korean, but also supports tate-chu-yoko (TCY Elements), which are blocks of horizontal text embedded in to vertical runs of text. The following scripts are supported:

- Latin (English, Spanish, French, Vietnamese, etc.)
- Greek, Cyrillic, Armenian, Georgian, Ethiopic
- Arabic, Hebrew
- Han ideographs and Kana (Chinese, Japanese & Korean) & Hangul Johab (Korean)
- Thai, Lao, Khmer
- Devanagari, Bengali, Gurmukhi, Malayalam, Telugu, Tamil, Gujarati, Oriya, Kannada, Tibetan
- Tifinagh, Yi, Cherokee, Canadian Syllabics, Deseret, Shavian, Vai, Tagalog, Hanunoo, Buhid, Tagbanwa

Advanced Typographic and Layout Features

Advanced Typographic support and Layout features include:

- Advanced text styling such as ligatures, typographic case, digit case, and digit width
- Advanced text layout control of kerning, tracking, leading, superscript, subscript, and baseline shift.
- Multiple columns of text, with each column considered a container of text
- Threaded text containers that support text that flows from one container to the next
- Inline graphics that are treated as characters within a text flow
- Support for tabs

For detailed descriptions and examples of the new text styling and text layout control features, see the Text Layout Component for Flash CS4 Professional Overview.

Flash Player 10 Required

The Text Layout Framework requires Flash Player 10 or later because the framework is built on Flash Player 10's new text engine (FTE). FTE can be accessed through the `flash.text.engine` package, which is part of the Flash Player 10 Application Programming Interface (API). The Flash Player API, however, provides very low-level access to the text engine, which means that some tasks can require a tremendous amount of code. When an API provides such low-level access, it is often helpful to have a software framework like the Text Layout Framework that not only encapsulates the low-level code into simpler APIs, but that also provides a conceptual architecture that organizes the basic building blocks defined by FTE into a system that is easy to use.

Unlike FTE, the Text Layout Framework is not built into Flash Player. Rather, it is an independent set of components written entirely in ActionScript 3.0 and is designed for use with both Flash CS4 and Flex Gumbo. Although the framework is designed for use with Flash CS4 and Flex Gumbo, it is not dependent on them. For example, if you use Flash CS4, there are two ways you can use the Text Layout Framework. If you prefer to drag-and-drop a text component on to the stage, you can use the new Flash CS4 TextLayout component that was created specifically for Flash CS4 based on the Text Layout Framework. Just to be clear, the TextLayout component is specific to Flash CS4 and is based on, but is not part of, the Text Layout Framework. If you prefer to use the Text Layout Framework directly to code in pure ActionScript 3.0, you can also do that in Flash CS4 as long as you ensure that the three components that comprise the Text Layout Framework are in your Flash CS4 library path. Similarly, if you use Flex Gumbo, you also have two options. You can either use the new FxTextArea component, which is based on the Text Layout Framework, or you can use the Text Layout Framework directly to code in pure ActionScript 3.0. Moreover, because Flex 3.2 supports Flash Player 10, you can even use the Text Layout Framework directly in Flex 3.2.

The Structure of the Text Layout Framework

The Text Layout Framework consists of three separate components:

- textLayout_core.swc
- textLayout_conversion.swc
- textLayout_edit.swc

The textLayout_core component (hereinafter the "core component") is the central component in the framework in that it handles the storage of text, the creation of text containers, and the display of text. You cannot use the Text Layout Framework without this core component. Not surprisingly, this component contains the majority of the code that comprises the framework.

The textLayout_conversion component (hereinafter the "conversion component") is used to import text into the framework and export text out of the framework. This component is necessary if you intend to use text that is not compiled directly into the SWF.

The textLayout_edit component (hereinafter the "edit component") is used to edit text that is stored in the core component. This component is necessary if you intend to make your text selectable or editable, or to allow your user to undo such editing.

All of the classes that comprise the framework can be found in subpackages of the `flashx.textLayout` package. For example, the classes that handle the storage of text can be found in the `flashx.textLayout.elements` package, whereas the classes that handle the creation of text containers can be found in the `flashx.textLayout.containers` package. All told, there are ten such subpackages, the names of which will make more sense to you as you learn more about the structure of the Text Layout Framework.

Architecture of the Text Layout Framework

To understand the architecture of the Text Layout Framework, it may help to understand the Model-View-Controller (MVC) design pattern. While the Text Layout Framework does not strictly adhere to the MVC pattern, the similarities are strong enough that understanding MVC will make it much easier to understand the architecture of the Text Layout Framework.

The MVC pattern calls for the separation of source code into modules that serve one of three purposes. The first module, the model, contains not only the raw data but also the set of rules that provide both structure and access to the raw data. The second module, the view, handles presentation of the data to the user. The third module, the controller, interprets interactions between the user and the view and translates user gestures, such as selecting or editing content, into commands to change data in the model. Generally there is only one model in a framework, but there can be several pairs of views and controllers.

The model for the Text Layout Framework is defined mainly by the elements package, which includes classes and interfaces that define the data structure that holds the text. The formats package is also part of the model because formatting information is stored as part of the model. The conversion class can also be considered part of the model in that it embodies the rules for how data is imported into and exported out of the model.

The view for the Text Layout Framework includes three packages that facilitate the rendering of text for display by Flash Player. The factory package provides a simple way to display static text. The container package includes classes and interfaces that define display containers for dynamic text. The compose package defines techniques for positioning and displaying dynamic text in containers.

The controller for the Text Layout Framework includes two packages that handle user interactions with the model. The edit and operations packages define classes that you can use to allow editing of text stored in the model.

The Text Layout Core Component

The core component embodies both the model and view of the Text Layout Framework. In this section we first examine the model's internal data structure and how text and text attributes are stored. Next, we take a closer look at the framework's view and introduce the flow composer, which composes text into container objects to render the text for viewing.

Text Flow Hierarchy

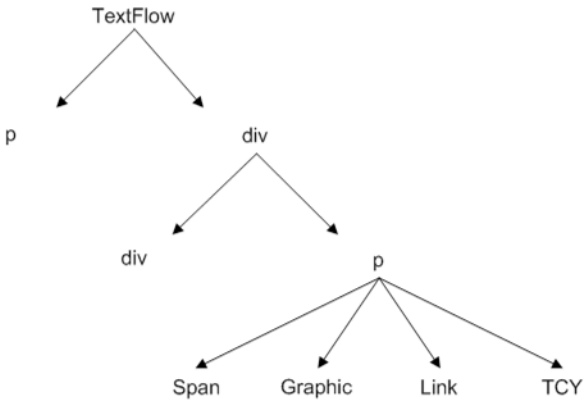
The model uses a hierarchical tree to represent text. Each node in the tree is an instance of a class defined in the elements package. For example, the root node of the tree is always an instance of the TextFlow class. The TextFlow class represents an entire story of text. The term story comes from page layout programs like PageMaker and InDesign, and refers to a collection of text that should be treated as one unit, or flow, even if the flow requires more than one column or text container to display.

Apart from the root node, the remaining elements are loosely based on XHTML elements. For example, the root node of the hierarchy can have children of only two types, which are defined by the classes DivElement and ParagraphElement. DivElements and ParagraphElements are similar, but not identical, to the and <p /> XHTML elements. DivElements are more narrowly defined than the XHTML element in that DivElements can contain only ParagraphElements and DivElements whereas the XHTML element can contain a wider variety of elements, including actual text. Fortunately, this difference is automatically handled for you during the import process. Text that is a direct child of a XHTML element is converted during import into a ParagraphElement with a child SpanElement that contains the text.

Like DivElements, ParagraphElements are also grouping elements that cannot directly contain text or graphics. ParagraphElements can, however, contain four types of child elements that do directly contain primitive text and graphics:

- The SpanElement class represents runs of text that share common formatting.
- The InlineGraphicElement class represents graphic elements that are treated as single characters in a line of text.
- The LinkElement class represents hypertext links, and are similar to <a /> XHTML elements. Links can contain one or more SpanElements, InlineGraphicElement or TCYElements.
- The TCYElement class represents short runs of text that are perpendicular to the rest of the line, usually small runs of horizontal text within vertical text lines. For example, you would use a TCYElement to represent a horizontal run of digits in an otherwise vertically oriented run of Japanese characters. A TCYElement can contain one or more SpanElements.

The hierarchy of the core component model is shown in the graphic below. Understanding this hierarchy is an important foundation for successful programming with the framework.



TextFlow Hierarchy

Understanding the structure of the core component model is also helpful when dealing with Text Layout Framework Markup, which is an XML representation of text that is part of the Text Layout Framework. Although the framework also supports other XML formats, the framework markup is unique in that it is based specifically on the structure of the TextFlow hierarchy. If you choose to export XML from a TextFlow using this markup format, the XML will be exported with this hierarchy intact.

You should also be aware that once you do start programming with the classes that define the nodes of this hierarchy, you will encounter a second hierarchy that relates these classes to one another. If you are familiar with Object Oriented Programming (OOP), you will recognize it as an inheritance hierarchy. All of the classes mentioned in this section derive—either directly or indirectly—from a class named FlowElement. This inheritance hierarchy is orthogonal to the model's text flow hierarchy. The two hierarchies intersect just enough to sow confusion in some, but in fact are largely independent.

Formats and FlowElements

You can assign formats to any FlowElement in the text flow hierarchy tree, from the root TextFlow instance down to the "leaves" (SpanElements, InlineGraphicElements, LinkElements, and TCYElements). The types of formatting that you can assign vary by context. For example, a format such as indent or margin naturally applies only to paragraphs, whereas a format such as font size would make sense to apply to individual characters or to paragraphs or even to the entire TextFlow. To make it easier to work with formats, the framework groups them into three categories: container, paragraph, and character.

Container formats apply exclusively to an entire container of text. Containers are discussed in the next section, but for now you can think of a container simply as a holder of text. Container attributes include settings such as padding values, column properties, line breaking and vertical alignment. You can apply container formats only to instances of the TextFlow, DivElement, and classes that implement the IContainerController interface, such as the DisplayObjectControllerContainer class. All of the container formats are conveniently bundled into the ContainerFormat class. There are two ways to apply container formatting. If you have a several formats to apply at one time, you can use the ContainerFormat class to create a special formatting object that contains all of the formatting values you want. You can then assign that object to the `containerFormat` property of any TextFlow, DivElement, or IContainerController object. Alternatively, you can set individual container formats on any TextFlow, DivElement or IContainerController. Each of these three classes or interfaces has a set of write-only properties that you can use to directly assign new container format values.

Paragraph formats are formats that apply to an entire paragraph of text, but that do not logically apply to individual characters. For example, paragraph formats include formats such as justification, margins, and tab stops. You can apply paragraph attributes only to instances of the ParagraphElement, DivElement, and TextFlow classes. Just as there is a ContainerFormat class for containers, there is also a ParagraphFormat class that you can use to apply paragraph formatting to paragraphs. You can assign an instance of the ParagraphFormat class to the `paragraphFormat` property of any ParagraphElement, DivElement, or TextFlow instance. Alternatively, you can set individual paragraph formats on any instance of ParagraphElement, DivElement or TextFlow by using the appropriate write-only property associated with that format.

Character formats are formats that apply to a single character or run of characters. Character formats include formats such as font size and color, tracking, kerning, and superscript. See the CharacterFormat class for a complete list of all character attributes. You can assign character formats to any FlowElement, which makes it easy to apply character formats to entire paragraphs or TextFlow instances. Character formats can be applied in the same two ways that paragraph formats can be applied. You can either create a CharacterFormat object and assign that object to any FlowElement's `characterFormat` property or you can assign individual character format values to any FlowElement instance by setting the appropriate write-only property on that instance.

Container, paragraph and character formats are inherited in accord with the TextFlow hierarchy. If you assign an instance of ContainerFormat to an entire TextFlow, every DisplayObjectContainerController in that TextFlow inherits the values of that instance. For example, if you set the padding value for a container at the TextFlow level, the setting applies to all containers in the TextFlow. You can, however, override the value in a given container by assigning a new value directly to the container.

If you assign an instance of ParagraphFormat to an entire TextFlow, every paragraph in that TextFlow inherits the values of that ParagraphFormat instance. For example, if you assign left and right margin values to an entire TextFlow, every paragraph in that TextFlow will inherit those values. But say that you later realize that one of the paragraphs in the text flow is a block quote and should be indented more than the other paragraphs. Fortunately, inherited attributes can be easily overridden by directly assigning a different instance of ParagraphFormat to a specific paragraph. To fix the indentation for the block quote, you need only create a new instance of ParagraphFormat that has new left and right margin values and assign that instance to the block quote paragraph. Attributes assigned directly to a FlowElement always take precedence over inherited attributes.

Character attributes are inherited in the same way. If you assign an instance of CharacterFormat to an entire TextFlow, every SpanElement, LinkElement, InlineGraphicElement, and TCYElement inherits those attributes. For example, if you want to use a specific font for all the text in your TextFlow, either assign that font value to an instance of CharacterFormat and then assign that CharacterFormat instance to the TextFlow or directly assign the font to the TextFlow's `fontFamily` property. All text in the entire TextFlow will be rendered with that font. As with paragraph attributes you can override the inherited value by assigning a new value directly to a SpanElement, LinkElement, etc.

There may also arise situations where you have assigned CharacterFormat values to an entire TextFlow, but you later decide that an entire paragraph should have different attribute values. Fortunately, you do not have to override the attributes for each individual FlowElement that is a child of the ParagraphElement. Instead, you can create a new instance of CharacterFormat and assign it to the ParagraphElement. All text in that paragraph will be rendered with the new attribute values because FlowElements always inherit attribute values from their most immediate ancestor. This feature makes it easier to manage formatting in separate parts of a TextFlow. For example, you can separate your TextFlow into two or more DivElements, each with a different set of formatting objects.

One thing to keep in mind when assigning formatting values is that assigning a formatting object to a FlowElement overrides any values that were previously set not only by a previous formatting object, but also by direct assignment. For example, if you directly assign the `fontSize` value of a SpanElement using the SpanElement's `fontSize` property, then later create a CharacterFormat object and assign it to the SpanElement's `characterFormat` property, the SpanElement takes on the `fontSize` value found in the CharacterFormat object. This is significant because every property of the CharacterFormat object has the value `null` by default, which means that you could inadvertently change the font size of your SpanElement. The following example shows an example of this scenario, in which the `fontSize` property is set to `null` by the assignment of a CharacterFormat object:

```
var span:SpanElement = new SpanElement();
span.text = "Hello, World";
span.fontSize = 48;
var ca:CharacterFormat = new CharacterFormat();
ca.fontFamily = "Helvetica";
span.characterFormat = ca;
```

The assignment of the CharacterFormat object resets the value of `fontSize` to null, which means that the value is now inherited from an ancestor element. To preserve the value of the `fontSize` property, send the SpanElement's existing CharacterFormat object as an argument to the CharacterFormat's constructor:

```
var span:SpanElement = new SpanElement();
span.text = "Hello, World";
span.fontSize = 48;
var cf:CharacterFormat = new CharacterFormat(span.characterFormat);
cf.fontFamily = "Helvetica";
span.characterFormat = cf;
```

Displaying Text

Now that we have seen how text is stored in the model, we turn to how the text is displayed by the view. Somehow, the text that is stored in the flow hierarchy must be converted into a format that Flash Player can display. The Text Layout Framework offers two ways to go about this—a simple approach suitable for displaying static text and a more complicated approach that allows for selection and editing of the text. In both cases, the text is ultimately converted into instances of the TextLine class, which is part of the new flash.text.engine package in Flash Player 10.

Rendering with TextLineFactory

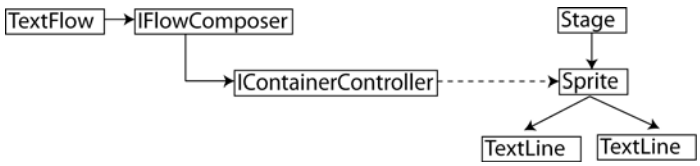
The simple approach uses the TextLineFactory class, which can be found in the flashx.textLayout.factory package. The advantage of this approach, beyond its simplicity, is that it has a smaller memory footprint than does the FlowComposer approach. This approach is advisable for static text that does not need to be edited or even selected and that fits within its display area without the need for scrolling.

The TextLineFactory class provides a static method named `createTextLinesFromTextFlow()` that converts the text from a TextFlow instance into a series of TextLine instances that are ready to be displayed. This approach involves a simple three step process. First, create a Rectangle instance to serve as the bounding box for the text. Second, create a callback function to be called after the `createTextLinesFromTextFlow()` method creates each TextLine instance. The callback function must add the TextLine instance to the Flash Player display list. Third, invoke the `creatTextLinesFromTextFlow()` method with three arguments: the name of your callback function, the name of your TextFlow instance, and the name of your bounding rectangle. The following example, which assumes that a TextFlow instance named `myFlow` already exists, demonstrates how this might look:

```
// assumes an existing TextFlow instance named myFlow
import flashx.textLayout.factory.TextLineFactory;
import flashx.textLayout.elements.TextFlow;
import flash.text.engine.TextLine;
import flash.geom.Rectangle;
// first, create a bounding rectangle
var bounds:Rectangle = new Rectangle(0,0,300,100);
// second, create a callback function
function callback(txLine:TextLine):void{    addChild(txLine); }
// third, call createTextLinesFromTextFlow()
TextLineFactory.createTextLinesFromTextFlow(callback, myFlow, bounds);
```

Rendering with Flow Composer

The simple approach may be all you need, but if you want to have more control over the display of the text, or if you want to give your user the ability to select and edit the text, you need to use something called a flow composer instead. A flow composer—which is an instance of the StandardFlowComposer class in the flashx.textLayout.compose package—is an object that manages not only the conversion of your TextFlow into TextLine instances, but also the placement of those TextLine instances into one or more containers.



An IFlowComposer has zero or more IContainerControllers

Every TextFlow instance has a corresponding object that implements the IFlowComposer interface. This IFlowComposer object is accessible through the `TextFlow.flowComposer` property. Through this property, you can call methods defined by the IFlowComposer interface that allow you to associate the text with one or more containers and prepare the text for display within a container.

Note: *Currently only one class in the framework, StandardFlowComposer, implements the IFlowComposer interface. This means that for now, the `TextFlow.flowComposer` property, is either `null` or points to an instance of StandardFlowComposer. In the future, there may be new classes added to the framework that also implement IFlowComposer.*

A container is simply an instance of the Sprite class, which is a subclass of the DisplayObjectContainer class. Both of these classes are part of the Flash Player display list API. If you are unfamiliar with the Flash Player display list, you can think of a DisplayObjectContainer as a container of objects that are in a format that Flash Player can render.

You can think of a container as a more advanced form of the bounding Rectangle used in the simple approach to displaying the TextFlow. Like the bounding Rectangle, a container circumscribes the area where TextLine instances will appear. Unlike a bounding Rectangle, a container can have scrolling enabled so that users can simply scroll downward to read text that does not fit into the container. Alternatively a container can be linked to another container such that text that would otherwise overflow the container simply appears in the linked container. Another advantage of using containers is that containers can have formatting applied in the same way that you apply formatting to a FlowElement. You can create an instance of the ContainerFormat class and apply it to a container. Moreover, a container can participate in the event handling mechanism and so play a part in handling user interactivity.

All of this enhanced functionality does come at the cost of some additional complexity. These extra attributes such as the formatting object and the scrolling options must be managed and stored. The framework handles this by wrapping each container in a controller object that comprises not only the container but also several properties related to formatting, scrolling and other container attributes. A controller object is an instance of the DisplayObjectContainerController class in the flashx.textLayout.container package. It is these controller objects, sometimes called “container controllers”, that are managed directly by the flow composer.

When you have your TextFlow populated and are ready to display it on screen, use the flow composer to create a controller object and associate it with the flow composer. Once you have the container associated, you must compose the text before it can be displayed. Accordingly, you can think of containers as having two stages: composition and display. Composing the text is the process of converting the text from the text flow hierarchy into TextLine instances and calculating how those instances will fit into the container. Displaying the text involves updating the Flash Player display list by calling the flow composer's `updateAllContainers()` method.

For example, the following code creates a TextFlow instance named `myFlow` and a controller object to display it. Once the controller object is created, it must be associated with the TextFlow’s `flowComposer` property. Finally, the call to the updateAllContainers() method causes the text to be composed and the display list to be update.

```
// import necessary classes
import flashx.textLayout.container.*;
import flashx.textLayout.compose.*;
import flashx.textLayout.elements.TextFlow;
import flashx.textLayout.conversion.TextFilter;

// first, create a TextFlow instance named myFlow
var markup:XML = <TextFlow><p><span>Hello, World</span></p></TextFlow>;
var textFlow:TextFlow = TextFilter.importToFlow(markup, TextFilter.TEXT_LAYOUT_FORMAT);

// second, create a controller
// the first parameter, this, must point to a DisplayObjectContainer
// the last two parameters set the initial size of the container in pixels
var contr:IContainerController = new DisplayObjectContainerController(this, 600, 600);

// third, associate it with the flowComposer property
myFlow.flowComposer.addController(contr);

// fourth, update the display list
myFlow.flowComposer.updateAllContainers();
```

The Text Layout Conversion Component

The conversion component allows you to import text into, and export text out of, the Text Layout Framework. You need to use this component if you plan to load text at runtime instead of compiling the text into the SWF or if you want to export text that is stored in a TextFlow instance into a String or XML object.

Both import and export are straightforward procedures. You call either the `export()` method or the `importToFlow()` method, both of which are part of the TextFilter class. Both methods are static, which means that you call the methods on the TextFilter class rather than on an instance of the TextFilter class.

The conversion component provides considerable flexibility in where you choose to store your text. For example, if you store your text in a database, you can import the text into the framework for display, use the editing component to allow changes to the text, and export the changed text back to your database.

Importing Text

The Text Layout Framework can import either plain text or XML in the form of Text Layout Markup.

To import plain text, specify that format with the second argument that you send to the `importToFlow()` method. In the following example, plain text is specified as the second argument to the `importToFlow()` method and the optional third argument is omitted:

```
var markup:String = "Hello World, this is plain text";
var flow:TextFlow = TextFilter.importToFlow(markup, TextFilter.PLAIN_TEXT_FORMAT);
```

Text Layout Markup is an XML representation of text that is stored in the Text Layout Framework. The XML mirrors the structure of the flow hierarchy and provides the highest fidelity representation of the text and formatting supported by the framework. To import text stored in Text Layout Markup, specify `TextFilter.TEXT_LAYOUT_FORMAT` as the second argument to the `importToFlow()` method, as shown in the following example:

```
var markup:XML = <TextFlow><p><span>Hello, World</span></p></TextFlow>;
var flow:TextFlow = TextFilter.importToFlow(markup, TextFilter.TEXT_LAYOUT_FORMAT);
```

Exporting Text

The Text Layout Framework can also export text to any of the three formats: plain text, FXG, or Text Layout Markup. To export from an existing TextFlow instance, call the `TextFilter.export()` method.

The `export()` method contains three required parameters and one optional parameter. The first parameter is the TextFlow instance from which you wish to export. The second parameter is which of the three formats—plain text, FXG, or Text Layout Markup—you want to apply to the exported text. The third parameter allows you to specify the data type of the exported text. You can choose either the type String or the type XML. The fourth and final parameter, which is optional, allows you to specify an instance of the ImportExportConfiguration class.

The following example imports a string in plain text format and exports it to an XML object in Text Layout Markup format:

```
var ptext:String = "Hello, World";
var flow:TextFlow = TextFilter.importToFlow(ptext, TextFilter.PLAIN_TEXT_FORMAT);
var out:XML = TextFilter.export(flow, TextFilter.TEXT_LAYOUT_FORMAT,
                                ConversionType.XML_TYPE );
```

Text Layout Framework Markup

Text Layout Framework Markup provides the highest fidelity representation of text in a TextFlow because the markup language provides not only tags for each of the TextFlow hierarchy's basic elements, but also attributes for all three types of formatting properties (ContainerFormat, ParagraphFormat, and CharacterFormat).

The following table contains the tags that can be used in Text Layout Framework Markup.

Element	Description	Formats	Children	Class
textflow	The root element of the markup.	container, paragraph, character	div, p	TextFlow
div	A division within a TextFlow. May contain a group of paragraphs.	container, paragraph, character	div, p	DivElement
p	A paragraph. May contain any of the elements listed in the rows below.	paragraph, character	a, tcy, span, img, tab, br	ParagraphElement
a	A link.	character	tcy, span, img, tab, br	LinkElement
tcy	A run of horizontal text (usually used in a vertical TextFlow).	character	a, span, img, tab, br	TCYElement
span	A run of text within a paragraph.	character		SpanElement
img	An image in a paragraph.	character		InlineGraphicElement
tab	A tab character.			TabElement
br	A break character. Used for ending a line within a paragraph; text will continue on the next line, but remain in the same paragraph.			BreakElement

The formatting properties, which can be found in the ContainerFormat, ParagraphFormat, and CharacterFormat classes, can be assigned directly to an element tag as an XML attribute. For example, in the following Text Layout Framework Markup the `fontSize` of the entire TextFlow is `14` and the color of the second paragraph is blue except for the last SpanElement, which is red because the color attribute in the last SpanElement overrides the color attribute of its parent in the hierarchy.

```
<flow:TextFlow xmlns:flow="http://ns.adobe.com/textLayout/2008" fontSize="14">
    <flow:p>This is an example of Text Layout Framework Markup.</flow:p>
    <flow:p color="#0000ff">
        <flow:span>This is the first span of the second paragraph.</flow:span>
        <flow:span color="#ff0000">This is the second span of the second
paragraph.</flow:span>
    </flow:p>
</flow:TextFlow>;
```

The Text Layout Edit Component

The MVC design pattern is particularly useful when discussing the Text Layout Edit Component because the edit component neatly embodies the functionality of the controller. The controller in the MVC design pattern encapsulates the code responsible for handling user interactivity with the view and translating user gestures into commands that modify the model.

The Edit component provides controller functionality by defining three classes that you can use to select text, edit text, or undo edits to the text. The SelectionManager class defines several dozen properties and methods you can use to manage the selection of text in your text flow. The EditManager class helps you manage requests to insert, delete, and format text. The UndoManager class allows you to maintain a history of the user's most recent editing activities and let the user undo or redo specific edits.

The ability to select or edit text is controlled at the TextFlow level. Every instance of the TextFlow class has an associated interaction manager, which is defined by the `TextFlow.interactionManager` property. To enable selection, assign an instance of the SelectionManager class to the `interactionManager` property. To enable both selection and editing, assign an instance of the EditManager class instead of an instance of the SelectionManager class.

The Selection Manager

To make your text selectable, create an instance of SelectionManager and associate it with the interaction manager of your TextFlow instance. For example, if you have a TextFlow instance named `flow` that you want to make selectable, create a SelectionManager object and assign it to the `flow` TextFlow instance's `interactionManager` property:

```
flow.interactionManager = new SelectionManager();
```

After a SelectionManager is assigned to a TextFlow's interaction manager, the TextFlow has access to the SelectionManager's event handlers. Through these event handlers, the SelectionManager can detect not only when text is selected or when the container gains or loses focus, but also when there is keyboard or mouse activity. The event handlers are methods of the SelectionManager class that can be overridden if you need custom event handling behavior.

The SelectionManager tracks selected text by managing text ranges. Theoretically, a text range can be thought of as a series of characters. The SelectionManager, however, uses a more efficient technique to track text ranges by taking advantage of the text flow hierarchy. Every character in the text flow hierarchy has an associated position that is relative to the start of the text flow. A character's absolute position is an integer that describes how many characters separate the current character from the start of the text flow. Using text positions, the SelectionManager can store text ranges as a pair of integers, one integer for the character closest to the start of the text flow, and another integer for the character farthest away from the start of the text flow. This is helpful for you to know because many of the properties and methods in SelectionManager refer to "an index into the text flow", which is equivalent to the absolute position, or distance from the start of the text flow. Text ranges are defined by the TextRange class, which happens to have a property named `absoluteStart` and another property named `absoluteEnd`. Both properties are integers that represent distances from the start of the text flow. Both carry the prefix "absolute" to signify that they represent distances from the very beginning of the text flow as opposed to the prefix "relative", which represents distances from an arbitrary FlowElement elsewhere in the text flow.

The Edit Manager

To enable selection and editing, create an instance of the EditManager class and associate it with the interaction manager of the TextFlow instance. For example, to enable editing for a TextFlow instance named `flow`, use the EditManager class:

```
flow.interactionManager = new EditManager();
```

The EditManager class is slightly more complex than the SelectionManager class because it must manage inserting, editing and formatting text. Rather than merely tracking selection ranges, the EditManager must manage a series of edits done by the user. To facilitate the management of these edits, the EditManager class creates a FlowOperation object to represent each edit. Every time a user initiates an insertion, edit, or formatting change, EditManager creates an instance of a FlowOperation subclass to represent that edit. The FlowOperation object encapsulates the code necessary not only to carry out the operation, but also to undo it. For example, if a user inserts text, EditManager creates an instance of the InsertTextOperation class, which inherits from the FlowOperation class.

Prior to executing an operation, EditManager dispatches an event object. Specifically, an instance of the FlowOperationEvent class, with its `type` property set to `FlowOperationEvent.FLOW_OPERATION_BEGIN`. This allows you to monitor when an operation is about to begin and to cancel the operation by calling `Event.preventDefault()`. This method is so named because an event can have an associated default behavior. For example, when text is inserted into a TextRange, an event object that represents the text insertion is dispatched before the text is actually inserted. That event object is said to have a default behavior that the text will be inserted into the TextRange because unless you take action to prevent that default behavior, the text will be inserted. There are a wide range of default behaviors associated with editing tasks such as text insertion, keyboard shortcuts, and menu selections like cut, copy and paste. All of these default behaviors can be prevented by calling the `Event.preventDefault()` method in your event handler function.

Because all operations generate the same type of event—a FlowOperationEvent instance of type `FLOW_OPERATION_BEGIN`—you will not know which operation is about to execute unless you check the `operation` property of the FlowOperationEvent class. The `operation` property stores a reference to the associated FlowOperation object.

After executing an operation, EditManager also dispatches an event object. The event object is also an instance of the FlowOperationEvent class, but its `type` property is set to `FlowOperationEvent.FLOW_OPERATION_END`. You can check whether an error was thrown during the execution of the operation by checking the `error` property of the FlowOperationEvent object. If an error occurs during the execution of an operation, EditManager stores a reference to that error in the `error` property of the event object before dispatching the event object. This gives you an opportunity to prevent Flash Player from throwing the error by calling `Event.PreventDefault()` on the FlowOperationEvent.

The Undo Manager

The UndoManager allows your user to undo and redo edits. Each EditManager has an associated UndoManager. To associate a specific UndoManager with an EditManager, create an instance of the UndoManager class and pass that instance as an argument to the EditManager constructor. This allows you to share a single UndoManager with multiple EditManagers, which means that a single UndoManager will manage multiple TextFlows. For example, to use the same UndoManager for two TextFlow instances named `flow1` and `flow2`, pass the same UndoManager instance as the sole argument to the EditManager constructors:

```
var undoMgr:UndoManager = new UndoManager();
flow1.interactionManager = new EditManager(undoMgr);
flow2.interactionManager = new EditManager(undoMgr);
```

If you create an EditManager without specifying an UndoManager, the framework automatically creates an UndoManager that is specific to that TextFlow. For example, in the following example, each flow has its own distinct UndoManager:

```
flow3.interactionManager = new EditManager();
flow4.interactionManager = new EditManager();
```

The UndoManager creates two stacks to track editing history. When an operation is executed the UndoManager pushes that operation onto the undo stack. If the user chooses to undo the operation, UndoManager removes, or pops, the operation from the undo stack and pushes it onto the redo stack. The size of the stacks can be configured with the UndoManager's `undoAndRedoItemLimit` property. If the size limit is reached, older operations are removed from the stack.