

This topic describes data binding, which lets you pass data between client-side objects in an Adobe Flex application. Binding automatically copies the value of a property of a source object to a property of a destination object when the source property changes. Binding lets you pass data between the different layers of the application, such as the user interface, data models, and data services.

Contents

| | |
|--|------|
| About data binding | 1203 |
| Data binding examples | 1208 |
| Binding to functions, Objects and arrays | 1211 |
| Using ActionScript in data binding expressions | 1221 |
| Using an E4X expression in a data binding expression | 1224 |
| Defining data bindings in ActionScript | 1228 |
| Using the Bindable metadata tag | 1231 |
| Considerations for using the binding feature | 1236 |

About data binding

Data binding is the process of tying the data in one object to another object. It provides a convenient way to pass data around in an application. Data binding requires a source property, a destination property, and a triggering event that indicates when to copy the data from the source to the destination. An object dispatches the triggering event when the source property changes.

Adobe Flex 2 provides three ways to specify data binding: the curly braces (`{}`) syntax in MXML, the `<mx:Binding>` tag in MXML, and the [BindingUtils](#) methods in ActionScript. The following example uses the curly braces (`{}`) syntax to show a Text control that gets its data from a TextInput control's text property:

```
<?xml version="1.0"?>
<!-- binding/BasicBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TextInput id="myTI"/>
    <mx:Text id="myText" text="{myTI.text}"/>
</mx:Application>
```

The property name inside the curly braces is the source property of the binding expression. When the value of the source property changes, Flex copies the current value of the source property, `myTI.text`, to the destination property, the Text control's text property.

You can include ActionScript code and E4X expressions as part of the data binding expressions, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/BasicBindingWithAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TextInput id="myTI"/>
    <mx:Text id="myText" text="{myTI.text.toUpperCase()}" />
</mx:Application>
```

In this example, you use the ActionScript method `String.toUpperCase()` to convert the text in the source property to upper case when Flex copies it to the destination property. For more information, see [“Using ActionScript in data binding expressions” on page 1221](#) and [“Using an E4X expression in a data binding expression” on page 1224](#).

You can use the `<mx:Binding>` tag as an alternative to the curly braces syntax. When you use the `<mx:Binding>` tag, you provide a source property in the `<mx:Binding>` tag's source property and a destination property in its destination property. The following example uses the `<mx:Binding>` tag to define a data binding from a TextInput control to a Text control:

```
<?xml version="1.0"?>
<!-- binding/BasicBindingMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TextInput id="myTI"/>
    <mx:Text id="myText"/>

    <mx:Binding source="myTI.text" destination="myText.text"/>
</mx:Application>
```

In contrast to the curly braces syntax, you can use the `<mx:Binding>` tag to completely separate the view (user interface) from the model. The `<mx:Binding>` tag also lets you bind multiple source properties to the same destination property because you can specify multiple `<mx:Binding>` tags with the same destination. For an example, see [“Binding more than one source property to a destination property” on page 1210](#).

The curly braces syntax and the `<mx:Binding>` tag both define a data binding at compile time. You can also use ActionScript code to define a data binding at run time, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/BasicBindingAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.binding.utils.*;

            // Define data binding.
            public function initBindingHandler():void {
                BindingUtils.bindProperty(myText, "text", myTI, "text");
            }
        ]]>
    </mx:Script>

    <mx:TextInput id="myTI"/>
    <mx:Text id="myText" preinitialize="initBindingHandler();"/>
</mx:Application>
```

In this example, you use the static `BindingUtils.bindProperty()` method to define the binding. You can also use the `BindingUtils.bindSetter()` method to define a binding to a function. For more information, see [“Defining data bindings in ActionScript” on page 1228](#).

Notice in this example that you use the `preinitialize` event to define the data binding. This is necessary because Flex triggers all data bindings at application startup when the source object dispatches the `initialize` event. For more information, see [“When data binding occurs” on page 1205](#).

When data binding occurs

Binding occurs under the following circumstances:

- The binding source dispatches an event because the source has been modified. This event can occur at any time during application execution. The event triggers Flex to copy the value of the source property to the destination property.
- At application startup when the source object dispatches the `initialize` event.

All data bindings are triggered once at application startup to initialize the destination property.

To monitor data binding, you can define a binding watcher that triggers an event handler when a data binding occurs. For more information, see [“Defining binding watchers” on page 1230](#).

The `executeBindings()` method of the `UIComponent` class executes all the bindings for which a `UIComponent` object is the destination. All containers and controls, as well as the `Repeater` component, extend the `UIComponent` class. The `executeChildBindings()` method of the `Container` and `Repeater` classes executes all of the bindings for which the child `UIComponent` components of a `Container` or `Repeater` class are destinations. All containers extend the `Container` class.

These methods give you a way to execute bindings that do not occur as expected. By adding one line of code, such as a call to `executeChildBindings()` method, you can update the user interface after making a change that does not cause bindings to execute. However, you should only use the `executeBindings()` method when you are sure that bindings do not execute automatically.

Properties that support data binding

You can use all properties of an object as the destination of a data binding expression. However, to use a property as the source of a data binding expression, the source object must be implemented to support data binding, which means that the object dispatches an event when the value of the property changes to trigger the binding. In this topic, a property that can be used as the source of a data-binding expression is referred to as a *bindable* property.

In the *Adobe Flex 2 Language Reference*, a property that can be used as the source of a data binding expression includes the following statement in its description:

“This property can be used as the source for data binding.”

For more information on creating properties that can be used as the source of a data binding expression, see [“Creating properties to use as the source for data binding” on page 1207](#).

Using read-only properties as the source for data binding

You can use a read-only property defined by a getter method, which means no setter method, as the source for a data-binding expression. Flex performs the data binding once when the application starts.

Using static properties as the source for data binding

You can automatically use a static constant as the source for a data-binding expression. Flex performs the data binding once when the application starts.

You can use a static variable as the source for a data-binding expression. Flex performs the data binding once when the application starts.

Creating properties to use as the source for data binding

When you create a property that you want to use as the source of a data binding expression, Flex can automatically copy the value of the source property to any destination property when the source property changes. To signal to Flex to perform the copy, you must use the `[Bindable]` metadata tag to register the property with Flex.

The `[Bindable]` metadata tag has the following syntax:

```
[Bindable]
[Bindable(event="eventname")]
```

If you omit the event name, Flex automatically creates an event named `propertyChange`, and Flex dispatches that event when the property changes to trigger any data bindings that use the property as a data-binding source. If you specify the event name, it is your responsibility to dispatch the event when the source property changes. For more information and examples of using the `[Bindable]` metadata tag, see [“Using the Bindable metadata tag” on page 1231](#).

The following example makes the `maxFontSize` and `minFontSize` properties that you defined as variables usable as the sources for data bindings expressions:

```
<?xml version="1.0"?>
<!-- binding/PropertyBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Define public vars for tracking font size.
            [Bindable]
            public var maxFontSize:Number = 15;

            [Bindable]
            public var minFontSize:Number = 5;
        ]]>
    </mx:Script>

    <mx:Text text="{maxFontSize}"/>
    <mx:Text text="{minFontSize}"/>

    <mx:Button click="maxFontSize=20; minFontSize=10;"/>
</mx:Application>
```

When you click the Button control, you update the values of the `maxFontSize` and `minFontSize` properties, and trigger a data binding update to the Text controls.

NOTE

If you omit the `[Bindable]` metadata tag, the Flex compiler issues a warning stating that the data binding mechanism cannot detect changes to the property.

Data binding uses

Common uses of data binding include the following:

- To bind properties of user interface controls to other user interface controls.
- To bind properties of user interface controls to a middle-tier data model, and to bind that data model's fields bound to a data service request (a three-tier system).
- To bind properties of user interface controls to data service requests.
- To bind data service results to properties of user interface controls.
- To bind data service results to a middle-tier data model, and to bind that data model's fields to user interface controls. For more information about data models, see [Chapter 39, "Storing Data,"](#) on page 1227.
- To bind an `ArrayCollection` or `XMLListCollection` object to the `dataProvider` property of a List-based control.
- To bind individual parts of complex properties to properties of user interface controls. An example would be a master-detail scenario in which clicking an item in a List control displays data in several other controls.
- To bind XML data to user interface controls by using ECMAScript for XML (E4X) expressions in binding expressions.

Although binding is a powerful mechanism, it is not appropriate for all situations. For example, for a complex user interface in which individual pieces must be updated based on strict timing, it would be preferable to use a method that assigns properties in order. Also, binding executes every time a property changes, so it is not the best solution when you want changes to be noticed only some of the time.

Data binding examples

This section contains common data binding examples.

Using data binding with data models

In the following example, a set of UI control properties act as the binding source for a data model. For more information about data models, see [Chapter 39, “Storing Data,” on page 1227](#).

```
<?xml version="1.0"?>
<!-- binding/BindingBraces.xml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Data model stores registration data that user enters. -->
    <mx:Model id="reg">
        <registration>
            <name>{fullname.text}</name>
            <email>{email.text}</email>
            <phone>{phone.text}</phone>
            <zip>{zip.text}</zip>
            <ssn>{ssn.text}</ssn>
        </registration>
    </mx:Model>

    <!-- Form contains user input controls. -->
    <mx:Form>
        <mx:FormItem label="Name" required="true">
            <mx:TextInput id="fullname" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Email" required="true">
            <mx:TextInput id="email" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Phone" required="true">
            <mx:TextInput id="phone" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Zip" required="true">
            <mx:TextInput id="zip" width="60"/>
        </mx:FormItem>

        <mx:FormItem label="Social Security" required="true">
            <mx:TextInput id="ssn" width="200"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Binding a source property to more than one destination property

You can bind a single source property to more than one destination property using the curly braces (`{}`) syntax, the `<mx:Binding>` tag, or the `BindingUtils` methods in ActionScript. In the following example, a `TextInput` control's `text` property is bound to properties of two data models, and the data model properties are bound to the `text` properties of two `Label` controls.

```
<?xml version="1.0"?>
<!-- binding/BindMultDestinations.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Model id="mod1">
        <data>
            <part>{input1.text}</part>
        </data>
    </mx:Model>

    <mx:Model id="mod2">
        <data>
            <part>{input1.text}</part>
        </data>
    </mx:Model>

    <mx:TextInput id="input1" text="Hello" />

    <mx:Label text="{mod1.part}"/>
    <mx:Label text="{mod2.part}"/>
</mx:Application>
```

Binding more than one source property to a destination property

You can bind more than one source property to the same destination property. You can set up one of these bindings using curly braces, but you must set up the others by using the `<mx:Binding>` tag, or by using calls to the `BindingUtils.bindProperty()` method or to the `BindingUtils.bindSetter()` method.

In the following example, the `TextArea` control is the binding destination, and both `input1.text` and `input2.text` are its binding sources:

```
<?xml version="1.0"?>
<!-- binding/BindMultSources.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Binding source="input2.text" destination="myTA.text"/>

    <mx:TextInput id="input1"/>
    <mx:TextInput id="input2"/>

    <mx:TextArea id="myTA" text="{input1.text}"/>
</mx:Application>
```

If `input1.text` or `input2.text` is updated, the `TextArea` control contains the updated value.

Defining bidirectional bindings

You can define a bidirectional data binding, where the source and destination of two data bindings are reversed, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/BindBiDirection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:TextInput id="input1" text="{input2.text}"/>
    <mx:TextInput id="input2" text="{input1.text}"/>
</mx:Application>
```

In this example, both `TextInput` controls are the source of a data binding expression, and both are a destination. When you modify `input1`, its value is copied to `input2`, and when you modify `input2`, its value is copied to `input1`.

Flex ensures that bidirectional data bindings do not result in an infinite loop; that is, Flex ensures that a bidirectional data binding is triggered only once when either source property is modified.

Binding to functions, Objects and arrays

This section describes how to bind to functions, Objects, and arrays.

Using functions as the source for a data binding

You can use a function as part of the source of a data binding expression. Two common techniques with functions are to use bindable properties as arguments to the function to trigger the function, or to trigger the function in response to a binding event. The following sections describe these techniques.

Using functions that take bindable properties as arguments

You can use ActionScript functions as the source of data binding expressions when using a bindable property as an argument of the function. When the bindable property changes, the function executes, and the result is written to the destination property, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/ASInBraces.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:CurrencyFormatter id="usdFormatter" precision="2"
        currencySymbol="$" alignSymbol="left"/>

    <mx:TextInput id="myTI" text="Enter number here"/>
    <mx:TextArea text="{usdFormatter.format(myTI.text)}/>
</mx:Application>
```

In this example, Flex calls the `CurrencyFormatter.format()` method to update the `TextArea` control every time the `text` property of the `TextInput` control is modified.

If the function is not passed an argument that can be used as the source of a data binding expression, the function only gets called once when the applications starts.

Binding to functions in response to a data-binding event

You can specify a function that takes no bindable arguments as the source of a data binding expression. However, you then need a way to invoke the function to update the destination of the data binding.

In the following example, you use the `[Bindable]` metadata tag to specify to Flex to invoke the `isEnabled()` function in response to the event `myFlagChanged`. When the `myFlag` setter gets called, it dispatches the `myFlagChanged` event to trigger any data bindings that use the `isEnabled()` function as the source:

```
<?xml version="1.0"?>
<!-- binding/ASFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import flash.events.Event;

            // Define a function that gets invoked
            // in response to the myFlagChanged event.
            [Bindable(event="myFlagChanged")]
            private function isEnabled():String {
                if (myFlag)
                    return 'true';
                else
                    return 'false';
            }

            private var _myFlag:Boolean = false;

            // Define a setter method that dispatches the
            // myFlagChanged event to trigger the data binding.
            public function set myFlag(value:Boolean):void {
                _myFlag = value;
                dispatchEvent(new Event("myFlagChanged"));
            }

            public function get myFlag():Boolean {
                return _myFlag;
            }
        ]]>
    </mx:Script>

    <!-- Use the function as the source of a data binding expression. -->
    <mx:TextArea id="myTA" text="{isEnabled()}" />

    <!-- Modify the property, causing the setter method to
         dispatch the myFlagChanged event to trigger data binding. -->
    <mx:Button label="Clear MyFlag" click="myFlag=false;" />
    <mx:Button label="Set MyFlag" click="myFlag=true;" />
</mx:Application>
```

For more information on the `[Bindable]` metadata tag, see [“Using the Bindable metadata tag” on page 1231](#).

Using data binding with Objects

When working with Objects, you have to consider when you define a binding to the Object, or when you define a binding to a property of the Object. This section describes both situations.

Binding to Objects

When you make an Object the source of a data binding expression, the data binding occurs when the Object is updated, or when a reference to the Object is updated, but not when an individual field of the Object is updated.

In the following example, you create subclass of Object that defines two properties, `stringProp` and `intProp`, but does not make the properties bindable:

```
package myComponents
{
    // binding/myComponents/NonBindableObject.as

    // Make no class properties bindable.
    public class NonBindableObject extends Object {

        public function NonBindableObject() {
            super();
        }

        public var stringProp:String = "String property";

        public var intProp:int = 52;
    }
}
```

Since the properties of the class are not bindable, Flex will not dispatch an event when they are updated to trigger data binding. You then use this class in a Flex application, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/WholeObjectBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initObj();">

    <mx:Script>
        <![CDATA[
            import myComponents.NonBindableObject;

            [Bindable]
            public var myObj:NonBindableObject = new NonBindableObject();

            [Bindable]
            public var anotherObj:NonBindableObject =
                new NonBindableObject();

            public function initObj():void {
                anotherObj.stringProp = 'anotherObject';
                anotherObj.intProp = 8;
            }
        ]]>
    </mx:Script>

    <!-- Data binding updated at application startup. -->
    <mx:Text id="text1" text="{myObj.stringProp}"/>

    <!-- Data binding updated at application startup. -->
    <mx:Text id="text2" text="{myObj.intProp}"/>

    <!-- Data bindings to stringProp not updated. -->
    <mx:Button label="Change myObj.stringProp"
        click="myObj.stringProp = 'new string';"/>

    <!-- Data bindings to intProp not updated. -->
    <mx:Button label="Change myObj.intProp"
        click="myObj.intProp = 10;"/>

    <!-- Data bindings to myObj and to myObj properties updated. -->
    <mx:Button label="Change myObj"
        click="myObj = anotherObj;"/>
</mx:Application>
```

Because you did not make the individual fields of the `NonBindableObject` class bindable, the data bindings for the two `Text` controls are updated at application start up, and when `myObj` is updated, but not when the individual properties of `myObj` are updated.

When you compile the application, the compiler outputs warning messages stating that the data binding mechanism will not be able to detect changes to `stringProp` and `intProp`.

Binding to properties of Objects

To make properties of an `Object` bindable, you create a new class definition for the `Object`, as the following example shows:

```
package myComponents
{
    // binding/myComponents/BindableObject.as

    // Make all class properties bindable.
    [Bindable]
    public class BindableObject extends Object {

        public function BindableObject() {
            super();
        }

        public var stringProp:String = "String property";

        public var intProp:int = 52;
    }
}
```

By placing the `[Bindable]` metadata tag before the class definition, you make bindable all public properties defined as variables, and all public properties defined by using both a setter and a getter method. You can then use the `stringProp` and `intProp` properties as the source for a data binding, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/SimpleObjectBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initObj();">

    <mx:Script>
        <![CDATA[
            import myComponents.BindableObject;

            [Bindable]
            public var myObj:BindableObject = new BindableObject();

            [Bindable]
            public var anotherObj:BindableObject =
                new BindableObject();

            public function initObj():void {
                anotherObj.stringProp = 'anotherObject';
                anotherObj.intProp = 8;
            }
        ]]>
    </mx:Script>

    <!-- Data binding updated at application startup. -->
    <mx:Text id="text1" text="{myObj.stringProp}"/>

    <!-- Data binding updated at application startup. -->
    <mx:Text id="text2" text="{myObj.intProp}"/>

    <!-- Data bindings to stringProp updated. -->
    <mx:Button label="Change myObj.stringProp"
        click="myObj.stringProp = 'new string';"/>

    <!-- Data bindings to intProp updated. -->
    <mx:Button label="Change myObj.intProp"
        click="myObj.intProp = 10;"/>

    <!-- Data bindings to myObj and to myObj properties updated. -->
    <mx:Button label="Change myObj"
        click="myObj = anotherObj;"/>
</mx:Application>
```

Binding with arrays

When working with arrays, such as `Array` or `ArrayCollection` objects, you can define the array as the source or destination of a data binding expression.

NOTE

When defining a data binding expression that uses an array as the source of a data binding expression, the array should be of type `ArrayCollection` because the `ArrayCollection` class dispatches an event when the array or the array elements change to trigger data binding. For example, a call to `ArrayCollection.addItem()`, `ArrayCollection.addItemAt()`, `ArrayCollection.removeItem()`, and `ArrayCollection.removeItemAt()` all trigger data binding. The `Array` class does not dispatch an event when it changes and, therefore, does not trigger data binding.

Binding to arrays

You often bind arrays to the `dataProvider` property of Flex controls, as the following example shows for the `List` control:

```
<?xml version="1.0"?>
<!-- binding/ArrayBindingDP.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var myAC:ArrayCollection = new ArrayCollection([
                "One", "Two", "Three", "Four"]);

            [Bindable]
            public var myAC2:ArrayCollection = new ArrayCollection([
                "Uno", "Dos", "Tres", "Quatro"]);
        ]]>
    </mx:Script>

    <!-- Data binding updated at application startup,
        when myAC is modified, and when an element of
        myAC is modified. -->
    <mx>List dataProvider="{myAC}"/>

    <!-- Data bindings to myAC updated. -->
    <mx:Button
        label="Change Element"
        click="myAC[0]='mod One'"/>

    <!-- Data bindings to myAC updated. -->
    <mx:Button
        label="Add Element"
        click="myAC.addItem('new element');"/>

    <!-- Data bindings to myAC updated. -->
    <mx:Button
        label="Remove Element 0"
        click="myAC.removeItemAt(0);"/>

    <!-- Data bindings to myAC updated. -->
    <mx:Button
        label="Change ArrayCollection"
        click="myAC=myAC2"/>
</mx:Application>
```

This example defines an `ArrayCollection` object, and then uses data binding to set the data provider of the `List` control to the `ArrayCollection`. When you modify an element of the `ArrayCollection` object or when you modify a reference to the `ArrayCollection` object, you trigger a data binding.

Binding to array elements

You can use individual `ArrayCollection` elements as the source or a binding expression, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/ArrayBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            [Bindable]
            public var myAC:ArrayCollection = new ArrayCollection([
                "One", "Two", "Three", "Four"]);

            [Bindable]
            public var myAC2:ArrayCollection = new ArrayCollection([
                "Uno", "Dos", "Tres", "Quatro"]);
        ]]>
    </mx:Script>

    <!-- Data binding updated at application startup
        and when myAC modified. -->
    <mx:Text id="text1" text="{myAC[0]}" />

    <!-- Data binding updated at application startup,
        when myAC modified, and when myAC[0] modified. -->
    <mx:Text id="text2" text="{myAC.getItemAt(0)}" />

    <mx:Button id="button1"
        label="Change Element"
        click="myAC[0]='new One'" />

    <mx:Button id="button2"
        label="Change ArrayCollection"
        click="myAC=myAC2" />
</mx:Application>
```

If you specify an array element as the source of a data binding expression using the square bracket syntax, [], data binding is only triggered when the application starts and when the array or a reference to the array is updated; data binding is not triggered when the individual array element is updated.

However, the data binding expression `myAC.getItemAt(0)` is triggered when an array element changes. Therefore, the `text2` Text control is updated when you click `button1`, while `text1` is not. When using an array element as the source of a data binding expression, you should use the `ArrayCollection.getItemAt()` method in the binding expression.

Clicking `button2` copies `myAC2` to `myAC`, and triggers all data bindings to array elements regardless of how you implemented them.

Using ActionScript in data binding expressions

This section describes how to use ActionScript in data binding expressions. You can use ActionScript in data binding expressions defined by curly braces and by the `<mx:Binding>` tag; however, you cannot use ActionScript when defining a data binding using the `BindingUtils.bindProperty()` or the `BindingUtils.bindSetter()` method.

Using ActionScript expressions in curly braces

Binding expressions in curly braces can contain an ActionScript expression that returns a value. For example, you can use the curly braces syntax for the following types of binding:

- A single bindable property inside curly braces
- To cast the data type of the source property to a type that matches the destination property
- String concatenation that includes a bindable property inside curly braces
- Calculations on a bindable property inside curly braces
- Conditional operations that evaluate a bindable property value

The following example shows a data model that uses each type of binding expression:

```
<?xml version="1.0"?>
<!-- binding/AsInBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Model id="myModel">
        <myModel>
            <!-- Perform simple property binding. -->
            <a>{nameInput.text}</a>
            <!-- Perform string concatenation. -->
            <b>This is {nameInput.text}</b>
            <!-- Perform a calculation. -->
            <c>{(Number(numberInput.text)) * 6 / 7}</c>
            <!-- Perform a conditional operation using a ternary operator. -->
            <d>{(isMale.selected) ? "Mr." : "Ms."} {nameInput.text}</d>
        </myModel>
    </mx:Model>

    <mx:Form>
        <mx:FormItem label="Last Name:">
            <mx:TextInput id="nameInput"/>
        </mx:FormItem>
        <mx:FormItem label="Select sex:">
            <mx:RadioButton id="isMale"
                label="Male"
                groupName="gender"
                selected="true"/>
            <mx:RadioButton id="isFemale"
                label="Female"
                groupName="gender"/>
        </mx:FormItem>
        <mx:FormItem label="Enter a number:">
            <mx:TextInput id="numberInput" text="0"/>
        </mx:FormItem>
    </mx:Form>

    <mx:Text
        text="'Calculation: '+numberInput.text+' * 6 / 7 = '+myModel.c'"/>
    <mx:Text text="'Conditional: '+myModel.d'"/>
</mx:Application>
```

Using ActionScript expressions in Binding tags

The `source` property of an `<mx:Binding>` tag can contain curly braces. When there are no curly braces in the `source` property, the value is treated as a single ActionScript expression. When there are curly braces in the `source` property, the value is treated as a concatenated ActionScript expression. The `<mx:Binding>` tags in the following example are valid and equivalent to each other:

```
<?xml version="1.0"?>
<!-- binding/ASInBindingTags.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public function whatDogAte():String {
                return "homework";
            }
        ]]>
    </mx:Script>

    <mx:Binding
        source="'The dog ate my '+ whatDogAte()"
        destination="field1.text"/>
    <mx:Binding
        source="{ 'The dog ate my '+ whatDogAte()}"
        destination="field2.text"/>
    <mx:Binding
        source="The dog ate my {whatDogAte()}"
        destination="field3.text"/>

    <mx:TextArea id="field1"/>
    <mx:TextArea id="field2"/>
    <mx:TextArea id="field3"/>
</mx:Application>
```

The `source` property in the following example is not valid because it is not an ActionScript expression:

```
<mx:Binding source="The dog ate my homework" destination="field1.text"/>
```

Using an E4X expression in a data binding expression

A binding expression in curly braces or an `<mx:Binding>` tag can contain an ECMAScript for XML (E4X) expression when the source of a binding is a bindable property of type XML. You cannot use E4X when defining a data binding using the `BindingUtils.bindProperty()` or the `BindingUtils.bindSetter()` method. This section describes how to use E4X expressions in data binding expressions.

Using an E4X expression in curly braces

A binding expression in curly braces automatically calls the `toString()` method when the binding destination is a `String` property. A binding expression in curly braces or an `<mx:Binding>` tag can contain an ECMAScript for XML (E4X) expression when the source of a binding is a bindable property of type XML; for more information, see [“Using an E4X expression in an `<mx:Binding>` tag” on page 1226](#).

In the code in the following example, there are three binding expressions in curly braces that bind data from an XML object. The first uses `.` (dot) notation, the second uses `..` (dot dot) notation, and the third uses `||` (or) notation.

```
<?xml version="1.0"?>
<!-- binding/E4XInBraces.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

                [Bindable]
                public var xdata:XML = <order>
                    <item id = "3456">
                        <description>Big Screen Television</description>
                        <price>1299.99</price><quantity>1</quantity>
                    </item>
                    <item id = "56789">
                        <description>DVD Player</description>
                        <price>399.99</price>
                        <quantity>1</quantity>
                    </item>
                </order>;
            ]]>
    </mx:Script>

    <mx:Label text="Using .. notation."/>
    <!-- Inline databinding will automatically call the
         toString() method when the binding destination is a string. -->
    <mx>List width="25%"
        dataProvider="{xdata..description}"/>

    <mx:Label text="Using . notation."/>
    <mx>List width="25%"
        dataProvider="{xdata.item.description}"/>

    <mx:Label text="Using || (or) notation."/>
    <mx>List width="25%"

    dataProvider="{xdata.item.@id=='3456' || @id=='56789'}.description}"/>
</mx:Application>
```

Using an E4X expression in an <mx:Binding> tag

Unlike an E4X expression in curly braces, when you use an E4X expression in an <mx:Binding> tag, you must explicitly call the `toString()` method when the binding destination is a `String` property.

In the code in the following example, there are three binding expressions in curly braces that bind data from an XML object. The first uses . (dot) notation, the second uses .. (dot dot) notation, and the third uses || (or) notation.

```
<?xml version="1.0"?>
<!-- binding/E4XInBindingTag.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600" height="900">

    <mx:Script>
        <![CDATA[
            [Bindable]
            public var xdata:XML =
                <order>
                    <item id = "3456">
                        <description>Big Screen Television</description>
                        <price>1299.99</price><quantity>1</quantity>
                    </item>
                    <item id = "56789">
                        <description>DVD Player</description>
                        <price>399.99</price>
                        <quantity>1</quantity>
                    </item>
                </order>;
        ]]>
    </mx:Script>

    <mx:Label text="Using .. notation."/>

    <!-- This will update because what is
        binded is actually the String and XMLList. -->
    <mx>List width="75%" id="txts"/>
    <mx:Binding
        source="xdata..description"
        destination="txts.dataProvider"/>

    <mx:Label text="Using . notation."/>
    <mx>List width="75%" id="txt2s"/>
    <mx:Binding
        source="xdata.item.description"
        destination="txt2s.dataProvider"/>

    <mx:Label text="Using || (or) notation."/>
    <mx>List width="75%" id="txt3s"/>
    <mx:Binding
        source="xdata.item.(@id=='3456' || @id=='56789').description"
        destination="txt3s.dataProvider"/>
</mx:Application>
```

Defining data bindings in ActionScript

You can define a data binding in ActionScript by using the [mx.binding.utils.BindingUtils](#) class. This class defines static methods that let you create a data binding to a property implemented as a variable, by using the `bindProperty()` method, or to a method, by using the `bindSetter()` method. For an example using the `bindProperty()` method, see “[Data binding examples](#)” on page 1208.

Differences between defining bindings in MXML and ActionScript

There are a few differences between defining data bindings in MXML at compile time and in defining them at runtime in ActionScript:

- You cannot include ActionScript code in a data binding expression defined by the `bindProperty()` or `bindSetter()` method. Instead, use the `bindSetter()` method to specify a method to call when the binding occurs.
- You cannot include an E4X expression in a data binding expression defined in ActionScript.
- You cannot include functions or array elements in property chains in a data binding expression defined by the `bindProperty()` or `bindSetter()` method. For more information on property chains, see “[Working with bindable property chains](#)” on page 1235.
- The MXML compiler has better warning and error detection support than runtime data bindings defined by the `bindProperty()` or `bindSetter()` method.

Example: Defining a data binding in ActionScript

The following example uses the `bindSetter()` method to set up a data binding. The arguments to the `bindSetter()` method specify the following:

- The source object
- The name of the source property.
- A method that is called when the source property changes

In the following example, as you enter text in the `TextInput` control, the text is converted to upper case as it is copied to the `TextArea` control:

```
<?xml version="1.0"?>
<!-- binding/BindSetterAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            import mx.binding.utils.*;
            import mx.events.FlexEvent;

            // Method called when myTI.text changes.
            public function updateMyString(val:String):void {
                myTA.text = val.toUpperCase();
            }

            <!-- Event listener to configure binding. -->
            public function mySetterBinding(event:FlexEvent):void {
                var watcherSetter:ChangeWatcher =
                    BindingUtils.bindSetter(updateMyString, myTI, "text");
            }
        ]]>
    </mx:Script>

    <mx:Label text="Bind Setter using setter method"/>
    <mx:TextInput id="myTI"
        text="Hello Setter" />
    <mx:TextArea id="myTA"
        initialize="mySetterBinding(event);"/>
</mx:Application>
```

Defining binding watchers

Flex includes the [mx.binding.utils.ChangeWatcher](#) class that you can use to define a data-binding watcher. Typically, a data-binding watcher invokes an event listener when a binding occurs. To set up a data-binding watcher, you use the static `watch()` method of the `ChangeWatcher` class, as the following example shows:

```
<?xml version="1.0"?>
<!-- binding/DetectWatcher.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initWatcher();">

    <mx:Script>
        <![CDATA[
            import mx.binding.utils.*;
            import mx.events.FlexEvent;
            import mx.events.PropertyChangeEvent;

            public var myWatcher:ChangeWatcher;

            // Define binding watcher.
            public function initWatcher():void {
                // Define a watcher for the text binding.
                ChangeWatcher.watch(textarea, "text", watcherListener);
            }

            // Event listener when binding occurs.
            public function watcherListener(event:Event):void {
                myTA1.text="binding occurred";

                // Use myWatcher.unwatch() to remove the watcher.
            }
        ]]>
    </mx:Script>

    <!-- Define a binding expression to watch. -->
    <mx:TextInput id="textinput" text="Hello"/>
    <mx:TextArea id="textarea" text="{textinput.text}"/>

    <!-- Trigger a binding. -->
    <mx:Button label="Submit" click="textinput.text='Goodbye';"/>
    <mx:TextArea id="myTA1"/>
</mx:Application>
```

You define an event listener for the data-binding watcher, where the event listener takes a single argument that contains the event object. The data type of the event object is determined by the property being watched. Each bindable property can dispatch a different event type and the associated event object. For more information on determining the event type, see [“Using the Bindable metadata tag” on page 1231](#).

Many event listener take an event object of type `PropertyChangeEvent` because that is the default data type of the event dispatched by bindable properties. You can always specify an event type of `flash.events.Event`, the base class for all Flex events.

Using the Bindable metadata tag

When a property is the source of a data binding expression, Flex automatically copies the value of the source property to any destination property when the source property changes. To signal to Flex to perform the copy, you must use the `[Bindable]` metadata tag to register the property with Flex, and the source property must dispatch an event.

The `[Bindable]` metadata tag has the following syntax:

```
[Bindable]
[Bindable(event="eventname")]
```

If you omit the event name, Flex automatically creates an event named `propertyChange`.

You can use the `[Bindable]` metadata tag in three places:

- Before a public class definition.

The `[Bindable]` metadata tag makes usable as the source of a binding expression all public properties that you defined as variables, and all public properties that are defined by using both a setter and a getter method. In this case, `[Bindable]` takes no parameters, as the following example shows:

```
[Bindable]
public class TextAreaFontControl extends TextArea {}
```

The Flex compiler automatically generates an event named `propertyChange` for all public properties so that the properties can be used as the source of a data binding expression. In this case, specifying the `[Bindable]` metadata tag with no event is the same as specifying the following:

```
[Bindable(event="propertyChange")]
```

If the property value remains the same on a write, Flex does not dispatch the event or update the property.

NOTE

When you use the `[Bindable]` metadata tag before a public class definition, it only applies to public properties; it does not apply to private or protected properties, or to properties defined in any other namespace. You must insert the `[Bindable]` metadata tag before a nonpublic property to make it usable as the source for a data binding expression.

- Before a public, protected, or private property defined as a variable to make that specific property support binding.

The tag can have the following forms:

```
[Bindable]
public var foo:String;
```

The Flex compiler automatically generates an event named `propertyChange` for the property. If the property value remains the same on a write, Flex does not dispatch the event or update the property.

You can also specify the event name, as the following example shows:

```
[Bindable(event="fooChanged")]
public var foo:String;
```

In this case, you are responsible for generating and dispatching the event, typically as part of some other method of your class. You can specify a `[Bindable]` tag that includes the event specification if you want to name the event, even when you already specified the `[Bindable]` tag at the class level.

- Before a public, protected, or private property defined by a getter or setter method. You must define both a setter and a getter method to use the `[Bindable]` tag with the property. If you define just a setter method, you create a write-only property that you cannot use as the source of a data-binding expression. If you define just a getter method, you create a read-only property that you can use as the source of a data-binding expression without inserting the `[Bindable]` metadata tag. This is similar to the way that you can use a variable, defined by using the `const` keyword, as the source for a data binding expression.

The tag can have the following forms:

```
[Bindable]
public function set shortNames(val:Boolean):void {
    ...
}

public function get shortNames():Boolean {
    ...
}
```

The Flex compiler automatically generates an event named `propertyChange` for the property. If the property value remains the same on a write, Flex does not dispatch the event or update the property. To determine if the property value changes, Flex calls the getter method to obtain the current value of the property.

You can specify the event name, as the following example shows:

```
[Bindable(event="changeShortNames")]
public function set shortNames(val:Boolean):void {
    ...
    // Create and dispatch event.
    dispatchEvent(new Event("changeShortNames"));
}

// Get method.
public function get shortNames():Boolean {
    ...
}
```

In this case, you are responsible for generating and dispatching the event, typically in the setter method, and Flex does not check to see if the old value and the new value are different. You can specify a `[Bindable]` tag that includes the event specification to name the event, even when you already specified the `[Bindable]` tag at the class level.

The following example makes the `maxFontSize` and `minFontSize` properties that you defined as variables that can be used as the sources for data bindings:

```
// Define public vars for tracking font size.
[Bindable]
public var maxFontSize:Number = 15;
[Bindable]
public var minFontSize:Number = 5;
```

In the following example, you make a public property that you defined by using a setter and a getter method that is usable as the source for data binding. The `[Bindable]` metadata tag includes the name of the event broadcast by the setter method when the property changes:

```
// Define private variable.
private var _maxFontSize:Number = 15;

[Bindable(event="maxFontSizeChanged")]
// Define public getter method.
public function get maxFontSize():Number {
    return _maxFontSize;
}

// Define public setter method.
public function set maxFontSize(value:Number):void {
    if (value <= 30) {
        _maxFontSize = value;
    } else _maxFontSize = 30;

    // Create event object.
    var eventObj:Event = new Event("maxFontSizeChanged");
    dispatchEvent(eventObj);
}
```

```
}
```

In this example, the setter updates the value of the property, and then creates and dispatches an event to invoke an update of the destination of the data binding.

In an MXML file, you can make all public properties that you defined as variables usable as the source for data binding by including the `[Bindable]` metadata tag in an `<mx:Metadata>` block, as the following example shows:

```
<mx:Metadata>
  [Bindable]
</mx:Metadata>
```

You can also use the `[Bindable]` metadata tag in an `<mx:Script>` block in an MXML file to make individual properties that you defined as variables usable as the source for a data binding expression. Alternatively, you can use the `[Bindable]` metadata tag with properties that you defined by using setter and getter methods.

Using read-only properties as the source for data binding

You can automatically use a read-only property defined by a getter method, which means no setter method, as the source for a data-binding expression. Flex performs the data binding once when the application starts.

Because the data binding from a read-only property occurs only once at application start up, you omit the `[Bindable]` metadata tag for the read-only property.

Using static properties as the source for data binding

You cannot use the `[Bindable]` metadata tag with a static variable. If you do, the compiler issues an error.

You can automatically use a static constant as the source for a data-binding expression. Flex performs the data binding once when the application starts. Because the data binding occurs only once at application start up, you omit the `[Bindable]` metadata tag for the static constant. The following example uses a static constant as the source for a data-binding expression:

```
<?xml version="1.0"?>
<!-- metadata/StaticBinding.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // This syntax casues a compiler error.
            // [Bindable]
            // public static var varString:String="A static var.";

            public static const constString:String="A static const.";
        ]]>
    </mx:Script>

    <!-- This binding occurs once at application startup. -->
    <mx:Button label="{constString}"/>
</mx:Application>
```

Working with bindable property chains

When you specify a property as the source of a data binding expression, Flex monitors not only that property for changes, but also the chain of properties leading up to it. The entire chain of properties, including the source property, is called a *bindable property chain*. In the following example, `firstName.text` is a bindable property chain that includes both a `firstName` object and its `text` property:

```
<mx:Text id="myText" text="{firstName.text}"/>
```

You can have a fairly long bindable property chain, as the following example shows:

```
<mx:Text id="myText" text="{user.name.firstName.text}"/>
```

For the data binding mechanism to detect changes to the `text` property, only the `text` property has to be bindable. However, if you want to assign a new value to any part of the chain at runtime, every element in the chain must be bindable. Otherwise, modifying the `user`, `name`, or `firstName` property at runtime results in the data binding mechanism no longer being able to detect changes to the `text` property.

When using the `BindingUtils.bindProperty()` or `BindingUtils.bindSetter()` method, you specify the bindable property chain as an argument to the method. For example, the `bindProperty()` method has the following signature:

```
public static function bindProperty(site:Object, prop:String, host:Object,
    chain:Object, commitOnly:Boolean = false):ChangeWatcher
```

The `host` and `chain` arguments specify the source of the data binding expression. You can define a data binding expression by using the `bindProperty()` method, as the following example shows:

```
bindProperty(myText, 'text', user, ["name", "firstName", "text"]);
```

In this example, `["name", "firstName", "text"]` defines the bindable property chain relative to the `user` object. Notice that `user` is not part of the bindable property change in this example.

In an MXML data-binding expression, the bindable property chain is always relative to `this`. Therefore, to define a data binding equivalent to the MXML data binding expression shown above, you write the `bindProperty()` method as the following example shows:

```
bindProperty(myText, 'text', this, ["user", "name", "firstName", "text"]);
```

Considerations for using the binding feature

Consider the following when using the binding feature:

- You cannot bind to style properties.
- If you bind a model into the `dataProvider` property of a component, you should not change items in the model directly. Instead, change the items through the Collections API. Otherwise, the component to which the model is bound is not redrawn to show the changes to the model. For example, instead of using the following:

```
myGrid.getItemAt(itemIndex).myField = 1;
```

You would use the following:

```
myGrid.dataProvider.editField(itemIndex, "myField", 1);
```

- Array elements cannot function as binding sources at run time. Arrays that are bound do not stay updated if individual fields of a source Array change. Binding copies values during instantiation after variables are declared in an `<mx:Script>` tag, but before event listeners execute.

Debugging data binding

In some situations, data binding may not appear to function correctly, and you may need to debug them. The following list contains suggestions for resolving data binding issues:

- Pay attention to warnings.

It is easy to see a warning and think that it doesn't matter, especially if binding appears to work at startup, but warnings are important.

If a warning is about a missing `[Bindable]` on a getter/setter property, even if the binding works at startup, subsequent changes to the property are not noticed.

If a warning is about a static variable or a built-in property, changes aren't noticed.
- Ensure that the source of the binding actually changed.

When your source is part of a larger procedure, it is easy to miss the fact that you never assigned the source.
- Ensure that the bindable event is being dispatched.

You can use the Flex command-line debugger (fdb), or the Adobe Flex Builder debugger to make sure that the `dispatchEvent()` method is called. Also, you can add a normal event listener to that class to make sure it gets called. If you want to add the event listener as a tag attribute, you must place the `[Event('myEvent')]` metadata at the top of your class definition or in an `<mx:Metadata>` tag in your MXML.
- Create a setter function and use a `<mx:Binding>` tag to assign into it.

You can then put a trace or an alert or some other debugging code in the setter with the value that is being assigned. This technique ensures that the binding itself is working. If the setter is called with the right information, you will know that it's your destination that is failing, and you can start debugging there.

