

Introduction to Adobe Flex 3

Trademarks

ChikaraDev and the ChikaraDev logo are trademarks of ChikaraDev. Such trademarks may be registered in the United States or in other jurisdictions, including internationally. This manual may include trademarks, service marks, or trade names of Adobe Systems Incorporated, Inc. and other companies. Such trademarks, service marks, or trade names may be registered in the United States or in other jurisdictions, including internationally.

Third-Party Information

This manual contains information such as links to third-party websites that are not under the control of ChikaraDev, and ChikaraDev is not responsible for the content on any linked site. If you access a third-party website mentioned in this manual, then you do so at your own risk. ChikaraDev has provided these links only as a convenience, and the inclusion of the link does not imply that ChikaraDev endorses or accepts any responsibility for the content on those third-party sites.

Copyright © 2008 ChikaraDev. All rights reserved.

The software described in this manual is provided under an agreement with Adobe Systems Incorporated, and such software can only be used in accordance with the terms of the agreement provided by Adobe Systems. Software code described and provided in this manual is provided under an agreement with ChikaraDev. The software can only be used in accordance with the terms of the agreement.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, photocopying, manual, optical, recording, or otherwise, outside the license agreement accompanying these materials, without the prior written permission of ChikaraDev. ChikaraDev claims copyright in this program and documentation as an unpublished work, revisions of which were first licensed on the date indicated in the foregoing notice. Claim of copyright does not imply waiver of other rights of ChikaraDev and its subsidiaries.

Information in this manual may change without notice and does not represent a commitment on the part of ChikaraDev.

NOTICE OF LIABILITY

This information in these training materials is distributed on an “AS IS” basis, without warranty of any kind, either express or implied. While every precaution has been taken in the preparation of these materials, neither ChikaraDev nor its licensors shall have any liability to any person or entity with respect to liability, loss, or damage caused or alleged to be caused directly or indirectly by the instructions contained in these materials or by the computer software and hardware products described herein.

First Edition: July 2008 - ChikaraDev - Cupertino, CA 95014 USA

What is Adobe Flex?

Adobe Flex is a technology providing an environment and infrastructure for creating rich internet applications. Adobe Flex Builder 3 is an IDE (integrated development environment) with integrated design, coding, and debugging tools that work together to help you work more efficiently.

When you create applications in Flex, you use primarily a combination of MXML, ActionScript, and CSS (cascading style sheets), though you could also incorporate HTML, JavaScript, and other technologies.

One of the greatest advantages of Adobe Flex over competing technologies, such as AJAX, Microsoft Silver light, and Java FX, is that Flex applications execute in the Adobe Flash Player browser plug-in. This eliminates most of the browser inconsistencies that you need to code for in some other technologies.

What is a RIA (rich internet application)?

Rich internet applications combine the richness more typically seen in desktop application user interfaces, with the low-cost development of web applications, together with engaging interactivity of multimedia communication.

In a nutshell, RIA are applications delivered over the internet that look better than your typical web pages, are more engaging for the user, and keep the user on your web site longer per session.

One key aspect of RIA is that unlike standard web pages where most user interaction results in a server request and complete page refresh, RIA execute more functionality on the client (end-user) browser, and only update that portion of the interface that changes due to user interaction.

Flex Applications == MXML, CSS, and ActionScript == MVC

You create Flex applications with three scripting “languages”, MXML, ActionScript, and CSS (cascading style sheets). Three are used to promote using the MVC (Model, View, Controller) design pattern.

Model components (ActionScript) - Encapsulates data and behaviors related to the data processed by the application. The model might represent the name, address, phone number and email address of a customer, or perhaps the contents of a shopping cart, or product descriptions and specifications.

View components (MXML and CSS) - Defines your application user interface, and the user's view of application data. The view might contain the Form container used to enter a customer's name, address, etc., a DataGrid control for displaying the contents of a shopping cart, or image of products in a catalog.

Controller components (ActionScript) - Handles data processing in your application. The controller provides application management and the business logic of the application. The controller does not necessarily have any knowledge of the view or the model.

What is MXML?

MXML is a tag based declarative language similar to XML and HTML. Using MXML tags, you can define Flex UI controls, such as `<mx:Button>`, `<mx:ComboBox>`, and `<mx:DataGrid>`. You can also use MXML to define non-UI Flex components, such as `<mx:HTTPService>` to retrieve and send data, `<mx:NumberFormatter>` to format data, and `<mx:XML>`, `<mx:ArrayCollection>`, and `<mx:XMLListCollection>` to create data elements.

Flex makes use of “properties” and “style properties” to set attributes of components. In some cases the distinction between properties and style properties is somewhat unclear, but in general properties relate to non-visual component attributes, and style properties are used to control the visual appearance of Flex UI components.

In MXML properties and style properties are set the same way within the opening MXML tag in the format **PROPERTY_NAME=“PROPERTY_VALUE”**. As we will see in the section on ActionScript, properties are set in ActionScript using “dot” notation common to many object oriented programming languages (`myObj.propName = propVal;`), but style properties are controlled using the `StyleManager.setStyle(“propName”, propVal)` and `getStyle(“propName”)` methods.

Many times MXML tags are defined with only an opening tag:

```
<mx:Label text="Enter your name:" fontSize="10"/>
```

In other instances opening and closing MXML tags are used to contain other tags within them.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:VBox width="400" height="300">
    <mx:Button id="myBtn" label="Click Me"/>
  </mx:VBox>
</mx:Application>
```

Starting with the top-level `<mx:Application>`, all the nested MXML tags in a Flex application make up what is known as the “**display list**”, a hierarchical set of “objects” making up the visual display of your application. When you define an MXML tag, the component it represents is automatically added to your application’s display list. Later we will see how adding components using ActionScript requires specifically adding the component to the display list using **`addChild()`**.

Two types of properties deserve special mention, **event** properties and **effect** properties.

Events - some properties are used to specify a function that is called when something “significant” happens to a Flex visual or non-visual, system component. Here are some event properties that can be set for the `<mx:Button>` control:

click – the button was clicked

hide – the “visible” property of the button was set to false

mouseOver – user moved the mouse over the button

```
<mx:Button label="Click Me" click="myClickListener(event);" hide="hideFunc(event);"/>
```

As we will see later, in ActionScript event listeners are specified with the `addEventListener()`.

Effects – effect properties specify an effect object that should be used when some significant event occurs. Here are some examples:

```
<mx:VBox hideEffect="{myFadeObj}" resizeEffect="{myResizeObj}" showEffect="{myFadeObj}"/>
```

The `id` property is very important in Flex. If you wish to refer to a Flex component property when setting a property in another Flex component, the referenced component should have an `id` property set.

```
<mx:CheckBox id="chbx" label="Ready to Go" selected="true"/>  
<mx:Button label="Click to Go" enabled="{chbx.selected}" click="goNow(event)"/>
```

Note: even if you wish to refer to a Flex component within that component’s tag, you still need an `id`. The following code will not give the desired result, setting the height of a `VBox` equal to the width:

```
<mx:VBox width="100" height="{width}" backgroundColor="0xFFFFFFFF"/>
```

Using the “`this`” keyword (used in many object oriented languages to refer to the “current object”) will not work, because “`this`” refers to either the root application tag or to the custom component root tag:

```
<mx:VBox width="100" height="{this.width}" backgroundColor="0xFFFFFFFF"/>
```

The above tag and the one before it without the “`this`” keyword will set the `VBox` height to the width of the application, not to the width of the `VBox`.

Here is how to achieve the desired result:

```
<mx:VBox id="myVB" width="100" height="{myVB.width}" backgroundColor="0xFFFFFF"/>
```

It is possible to code ActionScript within property values in MXML, though it is best to only do this if the ActionScript will be short. Here is an example of how to perhaps not use ActionScript in an MXML tag:

```
<mx:Label fontSize="20" text="Your average percentage based score, graded on a curve:"/>  
<mx:Label fontSize="20" text="{(score1+score2+score3/3)/300 * .4376}"/>
```

In this case it might be better to bind the Label text to a variable defined in ActionScript.

What is ActionScript?

ActionScript is a scripting language based on the ECMAScript standard on which JavaScript is based. ActionScript is used to define the Model and Controller of your application. JavaScript is an interpreted language, meaning it is processed as it is executed, whereas ActionScript is a compiled language, meaning it is processed (compiled) as you are writing your application, which leads to faster execution.

Much of your ActionScript code will consist of variables, functions, and classes, in addition to a variety of other ActionScript constructs for program control, such as conditionals, looping, etc. This section on ActionScript will provide an overview of variables, functions, classes and some of the other major programming constructs.

Variables - A variable is a named entity used to store some data, such as a number (49.99), a string ("Welcome Greg"), a Boolean value (true or false), a date (04/14/2009), etc. Variables correspond to properties and style properties in MXML.

To use a variable, you need to declare it and assign it a value. You can declare the variable and assign an initial value later, or you can assign a value when the variable is declared. Here are some examples:

```
var validuser:Boolean;  
var firstname:String = "Greg";  
var myVbox:VBox = new VBox();  
var currAdminPanel:AdminPanel = new AdminPanel();
```

Notice that a colon separates a variable from its "data type" (Boolean, String, AdminPanel).

Also notice how instances of classes (VBox and AdminPanel) are created above using the "new" keyword.

“Modifiers” can be added to variable declarations to affect aspects of the variables:

```
private var firstname:String = “Greg”;  
public static var num:int;  
protected const MULTIPLIER:uint = 50;
```

Each of these examples includes keywords that result in subtle variations in the qualities of the variables:

private, **public**, **protected**, and **internal** specify what parts of your code can use the variable.

var indicates the variable can change, and **const** indicates it is a constant, and once set cannot change.

static specifies a variable that has one value for all instances of a class. Without the static keyword each class instance has its own instance of the variable that can have a different value.

KEY POINT: you can declare and initialize variables outside functions, but you can only use and manipulate variables within functions (discussed next). So you can declare and initialize a variable num in a single statement outside a function, but you cannot assign it a value outside a function. If you are trying to use a variable outside a function and your statement does not include the “var” keyword, there is a good chance your statement will fail.

var num = 50; **okay – declaring and assigning a variable on the same line**

var num; **okay – declaring a variable**

num = 50; **error – cannot assign value to variable outside a function (unless declaring it)**

ActionScript Primitive Data Types

Here are the ActionScript “primitive” data types. All other Flex data types are objects.

String: a textual value, like a name or the text of a book chapter

Boolean: a true-or-false value, such as whether a switch is on or whether two values are equal

Numeric: ActionScript 3.0 includes three specific data types for numeric data:

Number: any numeric value, including values with or without a fraction

int: an integer (a whole number without a fraction)

uint: an "unsigned" integer, meaning a whole number that can't be negative

Functions - A function is used to do something, such as add two numbers, examine a string and report on its value, calculate the time span between two dates, respond to user interaction with you application, etc.

You define a function by specifying its “signature”. Here is a typical ActionScript function signature:

```
private function addNumbers(num1:uint, num2:uint):uint {  
    var result:uint = num1 + num2;  
    return result;  
}
```

Notice functions can be defined using “access modifiers” (private, public, protected) like variables to control what areas of your code can access the function.

The important parts and keywords of the function signature are:

function – this is what indicates this is a function, and not a variable (var) or a constant (const)

addNumbers – this is the name of the function

num1:uint – this is the first data parameter passed into the function for its use. num1 is the name of the parameter and uint is its data type.

uint after the closing parenthesis and colon is the data type of the value returned to the code that called the function. If the function does not return a value, this will be **void**. If a return value is specified, the function must return a value, and if the return data type is void, the function cannot return a value.

Statements that carry out the function’s work are enclosed in opening and closing curly braces { }. Functions do not need to accept parameters between the parenthesis, and if they accept them, the function does not need to make use of them in the function body enclosed in the { }.

The variable “result” declared within the addNumbers function above has a “scope” that is local to the function, and cannot be referenced outside the function curly braces { }. Keywords such as access modifiers private, public, protected and internal, and also other keywords such as static, etc. cannot be used inside functions.

The **...rest** parameter, allows you to pass any number of parameters, so in a function defined as **myFunction(... myParams)**, you refer to the parameters in the function body as array elements, myParams[0], myParams[1], myParams[3], and you can pass in any number of parameters.

```
private function myFancyFunction(... myArgs):void {  
    for (var i:uint = 0; i < myArgs.length; i++) {  
        trace(myArgs[i]);    // trace() prints data to the Flex Builder Console view  
    }  
}  
myFancyFunction (1, "Greg", true);
```

Output

```
1  
Greg  
true
```

As you can see, the parameters included in the ... rest parameter can be of any type, they don’t all have to be uints, Booleans, Strings, etc. The ... rest parameter can be used with other parameters, but it must be the last parameter listed.

```
private function myFancyFunction(num:uint, ... myArgs):void {
    trace(num);
    for (var i:uint = 0; i < myArgs.length; i++) {
        trace(myArgs[i]);
    }
}
myFancyFunction (400, 1, "Greg", true);
```

Output

```
400
1
Greg
true
```

All parameters passed into functions are passed in by “reference”, except for primitive values Boolean, int, Number, String, and uint. Passing by reference means you can pass as parameters data defined outside the function, and the data outside the function will be modified if the function changes the data. Primitives are passed by “value”, and data outside is not affected by modifications inside the function.

Classes – ActionScript is an object oriented language making use of many object oriented constructs, such as classes, packages, inheritance, accessor methods, method overriding, etc. Classes are such a large topic that only a brief overview of the topic will be covered here.

The following is a class we will use in our discussion of ActionScript classes.

```
package com.chikaradev.classes {
    import mx.controls.Button;

    public class MyButton extends Button{
        private var _myvar:int = 0;

        public function MyButton(){
            super();
        }

        public function set myvar(val:int):void{
            this._myvar = val;
        }

        public function get myvar():int{
            return this._myvar;
        }

        override protected function updateDisplayList(unscaledWidth:Number,
            unscaledHeight:Number):void{
            super.updateDisplayList(unscaledWidth, unscaledHeight);
            this.graphics.lineStyle(3, 0xFF0000, 1);
        }
    }
}
```

```

        this.graphics.moveTo(5, this.height/2);
        this.graphics.lineTo(this.width-5, this.height/2);
        validateNow();
    }
}
}

```

package com.chikaradev.classes – it is good to organize custom classes into packages that map to the directory structure where the classes are located, relative to the main application MXML file. In this case the file for custom class **MyButton** reside in the following directory structure:

```

src
  com
    chikaradev
      classes
        MyButton.as

```

import mx.controls.Button; - we need to import the Button class because our custom class “extends” Button, meaning it is based upon the Flex standard Button class.

public class MyButton extends Button – here we declare the custom class **MyButton**, with the access modifier **public**, using the **extends** keyword to indicate we are basing our class on the Flex **Button** class.

private var _myvar:int = 0; - our custom class will have a private variable **_myvar**. Often, the names of private variables in Flex start with an underscore.

public function MyButton() – this is the constructor of the class, where initialization occurs. We simply call the **super()** method, which calls the constructor of the “super” class, meaning the class this class extends (Button);

public function set myvar(val:int):void – Flex uses a feature called **set** and **get** methods to implement **accessor** methods, a key object oriented design construct. Implementing a set and get method here allows us to assign a value to **_myvar**, or to assess its value, using the **myvar="100"** syntax in the **<MyButton>** MXML tag. The method can be named anything, but in the body we assign the parameter to the private **_myvar** variable:

```

    this._myvar = val;

```

override protected function updateDisplayList – here we “**override**” the standard Flex Button class method **updateDisplayList**, meaning we want to change the standard behavior of the method. We first call the super class version of the **updateDisplayList** method, and then implement custom code, where we draw a line through the label of the button.

validateNow(); - often you want to call either **validateNow()** or **invalidateDisplayList()** after custom code that affects the display list, because your code may be executed but not rendered onscreen yet, and these methods force a redraw of the UI.

The following code illustrates defining an instance of our custom class in MXML and in ActionScript. Notice when you execute this simple application that the MyButton instance defined in MXML is displayed above the instance defined in ActionScript, because the MXML is processed before the creationComplete event is dispatched.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="init();" xmlns:comp="com.chikaradev.classes.*">
  <mx:Script>
    <![CDATA[
      import com.chikaradev.classes.MyButton;

      private function init():void{
        var mb:MyButton = new MyButton();
        mb.label = "Greg";
        mb.myvar = 35;
        this.addChild(mb);
      }
    ]]>
  </mx:Script>
  <comp:MyButton myvar="100" label="My New Button"/>
</mx:Application>

```

Array and ArrayCollection – arrays allow you to store multiple values in one data structure. You can create simple indexed arrays or associative arrays, known in some programming languages as hashes.

Note: the sample code in this section presents creating Arrays and ArrayCollection in ActionScript, but you can create them in MXML as well. See the Flex Builder help system for examples of creating Arrays and ArrayCollection in MXML.

You can declare an indexed array and assign it values at the same time:

```
var clients:Array = ["Bob", "Jim", "Susan"];
```

Or you can declare the array in one statement and assign values later:

```
var clients:Array = new Array();
...
clients[0] = "Bob";
clients[1] = "Jim";
clients[2] = "Susan";
```

Associative arrays (which are actually of the **Object** class, not the **Array** class) use “keys” instead of indexes to reference individual elements in the array:

```
var foodPrices:Object = {meat:"5.00", fish:"7.00", juice:"1.00"};
trace(foodPrices ["meat"], foodPrices ["fish"], foodPrices["juice"]);
```

You can also do this:

```
var foodPrices:Object = new Object();
foodPrices["meat"] = "5.00";
foodPrices["fish"] = "7.00";
foodPrices["juice"] = "1.00";
```

You can also use object oriented “dot” notation to refer to associative array elements:

```
foodPrices.meat = "5.00";
foodPrices.fish = "7.00";
foodPrices.juice = "1.00";
```

```
trace(foodPrices.meat, foodPrices.fish, foodPrices.juice);
```

ArrayCollection is a Flex class that wraps the Array class in order to provide additional functionality, such as the following methods not found in the Array class:

addItem(item:Object):void - adds the item to the end of the ArrayCollection.

addItemAt(item:Object, index:int):void - adds the item at the specified index.

contains(item:Object):Boolean - returns true if the ArrayCollection contains the specified object.

getItemAt(index:int, prefetch:int = 0):Object - gets the item at the specified index.

getItemIndex(item:Object):int - returns the index of the item if it is in the ArrayCollection.

refresh():Boolean - applies the sort and filter to the ArrayCollection.

removeAll():void - remove all items from the ArrayCollection.

Sometimes you need to check to see if two objects are the same object, often inside a function. You don't need to iterate the ArrayCollection, just do this:

```
if(myObject = myAC.getItemAt(myAC.getItemIndex(myObject))){
    Trace("Same object.");
}else{
    Trace("NOT same object.");
}
```

You can create an instance of an ArrayCollection like this:

```
private var myAC:ArrayCollection = new ArrayCollection([
    "meat", "fish", "juice", "cheese"
]);
```

You can create an ArrayCollection of objects like this:

```
public var foodsAC:ArrayCollection;
public function initData():void {
    foodsAC = new ArrayCollection(
        [{food:"meat", price:"5.00"},
        {food:"juice", price:"1.00"},
        {food:"cheese", price:"2.00"}
    ]);
}
```

XML, XMMList and XMMListCollection – similar to the relationship between the Array and ArrayCollection classes, the XML, XMMList, and XMMListCollection classes are related, and are used to process XML data.

Note: the sample code in this section presents creating XML, XMMList and XMMListCollections objects in ActionScript, but you can create them in MXML as well. See the Flex Builder help system for examples of creating XML, XMMList and XMMListCollections objects in MXML.

XML is a Flex class that represents XML data. An XML instance can be created like this:

```
var myXMLData:XML =
    <products>
        <item id='1'>
            <prodName>desk</ prodName>
            <price>129.99</price>
        </item>
        <item id='2'>
            <prodName>chair</ prodName>
            <price>59.99</price>
        </item>
    </products>
```

An XMMList instance represents an arbitrary collection of XML objects. Often an XMMList object is created while processing results returned from a Flex HTTPService request, which is often used to bring XML data into a Flex application.

The following code uses e4x syntax (**in bold**) to place XML data <item> elements into an XMMList.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="init();">
    <mx:Script>
        <![CDATA[
            private var productsXML:XMMList;
            private var myXMLData:XML =
                <products>
                    <item id='1'>
                        <prodName>desk</ prodName>
                        <price>129.99</price>
```

```

        </item>
        <item id='2'>
            <prodName>chair</ prodName>
            <price>59.99</price>
        </item>
    </products>

    private function init():void{
        productsXML = new XMMLList(myXMLData..item);
        for each(var xml:XML in productsXML){
            trace("****" + xml.toXMLString() + "****");
        }
    }
]]>
</mx:Script>
</mx:Application>

```

The XMMLListCollection class wraps the XMMLList class in order to provide additional functionality, such as the following methods not found in the XML and XMMLList classes:

addItem(item:Object):void - adds the item to the end of the XMMLListCollection.

addItemAt(item:Object, index:int):void - adds the item at the specified index.

getItemAt(index:int, prefetch:int = 0):Object - gets the item at the specified index.

getItemIndex(item:Object):int - returns the index of the item if it is in the XMMLListCollection.

refresh():Boolean - applies the sort and filter to the XMMLListCollection.

removeAll():void - remove all items from the XMMLListCollection.

Note: the XML, XMMLList, and XMMLListCollection classes contain many methods for accessing their data in different ways, and in different situations. The session of this training on processing XML data discusses many of these functions.

Note: one important difference between the XML, XMMLList, and XMMLListCollection classes is that while the XML and XMMLList classes can make use of e4x filtering syntax, the XMMLListCollection class cannot. The session of this training on processing XML data discusses using e4x syntax.

The following code adds to the previous example to create an XMMLListCollection from the XMMLList object, and uses trace() to print the second item:

```

import mx.collections.XMMLListCollection;
private var xlc:XMMLListCollection;
xlc = new XMMLListCollection(productsXML);
trace(xlc.getItemAt(1));

```

Conditionals – if, if/else, switch, and ternary conditional – Conditionals allow your ActionScript code to execute differently in various situations. ActionScript conditionals operate similar to in other programming languages.

if and if/else

Use **if** and **if/else** to execute code after performing a test that evaluates to a Boolean value:

```
private function init():void{
    var num:int = 100;
    if(num == 100){    // the test num == 100 is true
        trace("num is equal to 100.");
    }else{    // do this if the test evaluates to false
        trace("num is NOT equal to 100.");
    }
}
```

Note: non-zero numbers if(num) **also evaluate to true, but always use logical operators (==, !=, <, >=, etc.) for if/else tests.**

The **switch** conditional evaluates an expression and executes different code depending on the result.

```
private var num:int = 100;
private function checkNum(myNum:int):void{
    switch(myNum){
        case 100:
            trace("Number was 100.");
            break;
        case 200:
            trace("Number was 200.");
            break;
        case 300:
            trace("Number was 300.");
            break;
        default:
            trace("Number passed none of the tests.");
            break;
    }
}
```

break statements are necessary, otherwise in some situations more than one test could pass. The **default** block code (optional) would be executed if none of the other tests is true.

The **ternary** conditional is very concise and can be useful, but it should not be used if the resulting conditional expression is overly confusing.

The ternary conditional uses the **?** and **:** operators with the following generalized logic:

```
test ? trueVal : falseVal
```

```
num1 = num1 > 100 ? num2 : num1;
```

This expression is evaluated as:

If num1 > 100, assign value of num2 to num1, otherwise assign value of num1 to num1 (don't change it)

Looping – for, for..in, for each..in, while, do..while – The ActionScript **for**, **while**, and **do...while** looping mechanisms operate similar to other programming languages. **for..in** and **for each..in** are somewhat more unique.

A **for** loop executes a predetermined number of times by initializing a value, testing that value against some other value, and then changing the value. The **for** loop executes until the test evaluates to false.

```
for(var cnt:uint = 0; cnt < 5; cnt++){    // you could also decrement cnt    cnt--
    trace("cnt: " + cnt);
}
```

Output

```
cnt: 0
cnt: 1
cnt: 2
cnt: 3
cnt: 4
```

A **while** loop executes until a test evaluates to false. You need to make sure you update the test variable in the **while** loop, otherwise you could end up with an infinite loop.

```
var cnt:uint = 0;
while(cnt < 5){
    trace("cnt: " + cnt);
    cnt++;    // don't forget to update cnt so eventually test is true, or you get an infinite loop
}
```

Output

```
cnt: 0
cnt: 1
cnt: 2
cnt: 3
cnt: 4
```

A **do..while** loop executes like a **while** loop, but it is guaranteed to execute at least once, even if the test evaluates to false the first time. You need to make sure you update the test variable in the **do..while** loop, otherwise you could end up with an infinite loop.

```
var cnt:uint = 0;
do{
    trace("cnt: " + cnt);
    cnt++;    // don't forget to update cnt so eventually test is true, or you get an infinite loop
} while(cnt < 5);
```

Output

```
cnt: 0
cnt: 1
cnt: 2
cnt: 3
cnt: 4
```

A **for..in** loop iterates through the properties of an object, or the elements of an array.

```
var myObj:Object = {num1:20, num2:30};
for (var n:String in myObj){
    trace(n + ": " + myObj[n]);
}
```

Output

```
num2: 30
num1: 20
```

```
var myArray:Array = ["one", "two", "three"];
for (var m:String in myArray){
    trace(myArray[m]);
}
```

Output

```
one
two
three
```

The **for each..in** loop iterates through the items of an Object or a collection, such as tags in an XML or XMLList object, or the elements of an array. Unlike the **for..in** loop, the iterator variable in a **for each..in** loop contains the value held by the property instead of the name of the property, so if you want to get the actual keys of an associative array, use the **for..in** loop, not the **for each..in** loop.

```
var myObj:Object = {num1:20, num2:30};
for each (var num:Object in myObj){
    trace(num);
}
```

Output

```
30
20
```

```
var myXML:XML = <users>
    <fname>George</fname>
    <fname>Harry</fname>
    <fname>Lisa</fname>
</users>;
```

```
for each (var name:String in myXML.fname){  
    trace(name);  
}
```

Output

George
Harry
Lisa

```
var myArray:Array = ["one", "two", "three"];  
for each (var item:String in myArray){  
    trace(item);  
}
```

Output

one
two
three