Flash ActionScript 2.0 Overview

Flash ActionScript 2.0 overview

ActionScript is a programming language similar to the JavaScript programming language. ActionScript is the language you use to add interactivity to Flash applications, whether your applications are simple animated movies or more complex applications.

The ActionScript language has grown and developed since its introduction several years ago. Flash introduces new language elements that implement object-oriented programming. Because these language elements represent a significant enhancement to the previous ActionScript language, they represent a new version of ActionScript: ActionScript 2.0.

➔ Applications developed with ActionScript 2.0 require Flash Player 6 or later.

The default Publish setting for new files created in Flash is ActionScript 2.0. If you plan to modify an existing FLA file to use ActionScript 2.0 syntax, ensure that the FLA file specifies ActionScript 2.0 in its Publish settings. If it does not, your file **will compile incorrectly**, although Flash will not generate compiler errors.

Terminology

As with all scripting languages, ActionScript uses its own terminology. The following list provides an introduction to some ActionScript terms.

Events

Events occur during the playback of a movie. There are predefined events that the Flash player gives you access to, such as the act of moving or clicking the mouse *on(mouseDown)*. Every script is triggered by an event, and your movie can react to numerous events, everything from a button being released to text changing in a text field to a sound completing its playback, and more.

```
on(mouseDown) {
    // your script goes here
}
```

Operators

Operators are symbols that calculate a new value from one or more values. For example, the addition (+) operator adds two or more values together to produce a new value. Operators include a number of symbols (=, <, >, +, -, *, &&, etc.) and are used to connect two elements in a script in various ways. For example:

```
var taxPercent:Number = .06;
```

- assigns a numeric value of .06 to the variable named *taxPercent* using the assignment operator, =.

```
amountA < amountB;
```

- evaluates if *amountA* is less than *amountB*.

```
value1 * 500;
```

- multiplies *value1* times 500.

The operators >, <, >=, <=, ==, !=, ===, are known as comparison operators.

Keywords

Keywords are words reserved for specific purposes within ActionScript syntax. As such, they cannot be used as variable, function, or label names. For example, the word on is a keyword and can only be used in a script to denote an event that triggers a script, such as *on (press)*, *on (rollOver)*, *on (rollOut)*, and so forth. Attempting to use keywords in your scripts for anything other than their intended purpose will result in errors. Here are some other keywords: *break, case, class, continue, default, delete, do, dynamic, else, extends, finally, for, function, get, if, implements, import, interface, in, instanceof, new, null, private, public, return, set, static, switch, this, throw, try, typeof, undefined, var, void, while,* and *with*.

Classes

"Classes of objects", or "object classes", or simply "classes", are phrases used to describe sets of objects with similar characteristics and behaviours. The terms upper class, middle class, or working class describe groups of people that fit a certain belonging due to their finances.

In the real world, animals belong to the animal class, persons belong to the person class, and cars belong to the car class.

♦ **Objects**

Objects are the key to understanding object oriented programming. In object oriented programming, an object is just as real as an object in the real world. For example, a dog, or a car are objects that exist in the real world.

Objects are made up of properties and methods. Each object is an **instance** of a particular class, has its own name, properties and methods.

Real world objects share two characteristics with objects in the computer world. They are:

♦ Properties

Properties are attributes (characteristics) that describe an object. For example, dogs have properties such as their name, colour, breed, and if they are hungry.

♦ Methods

Methods describe an object's behaviours. Dog behaviours would be barking, fetching, and wagging their tails.

Computer objects are modelled after real world objects in that they also have specific properties and behaviour.

Syntax rules

As with all programming languages, ActionScript has syntax rules that you must follow to create scripts that can compile and run correctly.

Case sensitivity

ActionScript 2.0 is a **case sensitive** language, which means variable names that differ in case, such as **book** and **`Book`**, are considered different from each other. Therefore, it is important to follow consistent capitalisation conventions, to make it easy to identify names of functions and variables in ActionScript code.

When you publish files for Flash Player 7 or later, Flash **implements case sensitivity** whether you are using ActionScript 1 or ActionScript 2.0. This means that keywords, class names, variables, method names, etc are all case sensitive.

Dot Syntax

Dots (`.`) are used within scripts in a couple of ways. One is to denote the target path to a specific Timeline. For example,

```
_root.myCar_mc.frontWheel_mc
```

- points to a movie clip on the main (*_root*) Timeline named *myCar_mc*, which contains a movie clip named *frontWheel_mc*.

Because ActionScript is an object-oriented language, most interactive tasks are accomplished by changing a property (characteristic) of an object or by telling an object to do something (invoking a method). When changing a property or when invoking a method, dots are used to separate the object's name from the property or method being worked with. For example, movie clips are objects; to set the rotation property of a movie clip instance named *wheel_mc*, you would use the following syntax:

```
wheel_mc._rotation = 45;
```

Notice that a dot separates the name of the object from the property being set.

To tell the same movie clip instance to play, invoking the *play()* method, you would use this syntax:

```
wheel_mc.play();
```

Once again, a dot separates the name of the object from the method invoked.

Curly { } brackets

ActionScript event handlers, class definitions, and functions are grouped together into blocks of code by using curly brackets { }. Generally, anything between opening and closing curly brackets signifies an action or set of actions the script needs to perform when triggered. Think of curly brackets as saying, "As a result of this {do this}." For example:

```
on (release) {
  //set the cost of the mug
  var mugCost:Number = 5;
  //set the local sales tax percentage
  var taxPercent:Number = .06;
```

```
    }
```

The preceding could be viewed as a sentence that says this: As a result of releasing the button, create two variables called *mugCost* and *taxPercent*.

You can put the opening bracket on the same line as your declaration or on the next line.

```
  for (var i=1; i<=10; i++){
    sum += i;
  }
```

or

```
  for (var i=1; i<=10; i++)
  {
    sum += i;
  }
```

You can check for matching curly brackets in your scripts using syntax and punctuation checking.

Parentheses ( )

Parentheses are used in various ways in ActionScript. For the most part, scripts employ parentheses to set a specific value that an action will use during its execution. The sample script that tells the *myCarr_mc* movie clip instance to go to and play Frame 50:

```
      myCar_mc.gotoAndPlay (50);
```

If the value within parentheses is changed from 50 to 20, the action still performs the same basic task (moving the *myCar_mc* movie clip instance to a specified frame number). It just does so according to the new value. Parentheses are a way of telling an action to work based on what is specified between the parentheses.

Quotation Marks "

Quotation marks are used to denote textual data in the script, called a string. Because text is used in the actual creation of the script, quotation marks provide the only means for a script to distinguish between instructions (pieces of data) and actual words. For example, *Kelly* (without quotes) signifies the name of a piece of data. On the other hand, *"Kelly"* signifies the actual word "Kelly."

```
      myCar_mc.gotoAndPlay ("racing");
```

- this action moves the movie clip *myCar_mc* to a label named *racing*.

Semicolons ;

An ActionScript statement is terminated with a semicolon, as shown in the following examples:

```
  var row:Number = 0;
```

If you omit the terminating semicolon, Flash still compiles your script successfully. However, it is good scripting practice to use semicolons because it makes your code more readable.

Semicolons are **required within** *for* **loops**, as shown in the following example:

```
//a for loop that adds numbers 1-10
var sum:Number = 0;
for (var i=1; i<=10; i++) {
  sum += i;
}
```

Comments //

Comments are useful for tracking what you intended to do and for passing information to other developers if you work in a collaborative environment or are providing samples. Even a simple script is easier to understand if you make notes as you create it.

To indicate that a line or portion of a line is a comment, precede the comment with two forward slashes (`//`):

```
//a for loop that adds numbers 1-10

for (var i=1; i<=10; i++) {//a for loop
```

Comments can be of any length without affecting the size of the compiled file, and they do not need to follow rules for ActionScript syntax or keywords.

You can also create multi-line comments using the following syntax:

```
/* everything between
here is considered
a comment */
```

Indenting, spacing

Although not absolutely necessary, it is a good idea to indent and space the syntax in your code. For example, the following:

```
on (release) {
var mugCost:Number = 5;
}
```

executes the same way as this:

```
on (release) {
  var mugCost:Number = 5;
}
```

However, by indenting the second line of code, it is easier to read. A good rule is to indent anything within curly braces to indicate that the code within those braces represents a code block or chunk of code that is to be executed at the same time. The AutoFormat feature of the Actions panel takes care of most of this formatting.

Data types

A data type describes a piece of data and the kinds of operations that can be performed on it. That data is stored in a variable. You use data types when creating variables, object instances, and function definitions.

String data type

A string is a sequence of characters such as letters, numbers, and punctuation marks. You enter strings in an ActionScript statement by enclosing them in double (") quotation marks.

A common way to use the string type is to assign a string to a variable. For example, in the following statement, `"L7"` is a string assigned to the variable

```
     var favouriteBand_str:String = "U2";
```

You can use the addition (+) operator to concatenate or join two strings. The following expression includes a space after the comma:

```
  var greeting_str:String = "Welcome, " + firstName;
```

To include a quotation mark in a string, precede it with a backslash character (\). This is called escaping a character. There are other characters that cannot be represented in ActionScript except by special escape sequences.

Number data type

The number data type is a double-precision floating-point number. Numeric operators add, subtract, multiply, divide, and perform other arithmetic operations.

Operator	**Operation performed**
+	Addition
*	Multiplication
/	Division
%	Modulo (remainder of division)
-	Subtraction
++	Increment
--	Decrement

The most common use of the increment operator is i++ instead of the more verbose i = i+1. You can use the increment operator before or after an operand. In the following example, *age* is incremented first and then tested against the number 30:

```
  if (++age >= 30)
```

This process is known as a pre-increment. In the following example, *age* is incremented after the test is performed:

```
  if (age++ >= 30)
```

This process is known as a post-increment.

Boolean data type

A Boolean value is one that is either *true* or *false*. ActionScript converts the values to 1 and 0 when appropriate. Boolean values are most often used with logical operators in ActionScript statements that make comparisons to control the flow of a script.

Object data type

An object is a collection of properties. Each property has a name and a value. The value of a property can be any Flash data type. This lets you arrange objects inside each other, or *nest* them. To specify objects and their properties, you use the dot (.) operator. For example, in the following code, *hoursWorked* is a property of *weeklyStats*, which is a property of *employee*:

```
    employee.weeklyStats.hoursWorked
```

The ActionScript MovieClip object has methods that let you control movie clip symbol instances on the Stage. This example uses the *play()* and *nextFrame()* methods:

```
  mcInstanceName.play();
  mc2InstanceName.nextFrame();
```

movieClip data type

Movie clips are symbols that can play animation in a Flash application. They are the data type that refers to a graphic element.

Null data type

The null data type has only one value, *null*. This value means no value, that is, a lack of data. The following example demonstrates how you can use *null* to test if form fields currently have form focus:

```
  if (Selection.getFocus() == null) {
    trace("no selection");
  }
```

Undefined data type

The undefined data type has one value, *undefined*, and is automatically assigned to a variable to which a value has not been assigned, either by code or user interaction.

The value *undefined* is automatically assigned. You use the undefined data type to check if a variable is set or defined. This data type lets you write code that executes only when the application is running, as shown in the following example:

```
  if (init == undefined) {
    trace("initializing app");
    init = true;
  }
```

If your application has multiple frames, the code does not execute a second time because the *init* variable is no longer undefined.

Void data type

The void data type has one value, *void*, and is used in a function definition to indicate that the function does not return a value, as shown in the following example:

```
  //Creates a function with a return type Void
  function displayFromURL(url:String):Void
```

Converting data types

Converting numeric to a string (ASCII):

```
_root.score_txt.text = String(totalScore );
```

Converting strings to a number:

```
Number("468") returns 468
Number("23.45") returns 23.45
parseInt("13.3") returns 13
parseFloat("13.3") returns 13.3
```

Variables

A variable is like a container in computer memory that is given a unique name and holds a value, for example, a number or text. Using ActionScript, the content of variables can be created, changed, and retrieved. **Data types** describe the kind of information a variable can contain. The ActionScript data types are, for example, string, number, Boolean, object, movieClip, function, null, and undefined.

Variable names **are case-sensitive**: *firstname* and *firstName* are not the same.

In the following example, the variables on the left side are assigned different types of values:

```
var x:Number = 5;
var name:String = "Lulu";
var myColour:Color = new Color(mcInstanceName);
```

It is a good idea always to assign a variable a known value the first time you define the variable. This is known as *initialising* a variable and is often done in the first frame of the SWF file.

Trace()

To view the value of a variable, you use the *trace()* statement to send the value to the Output window. For example, the action

```
trace(hoursWorked)
trace("game over " + duration)
-displays the text
```

sends the value of the variable *hoursWorked* to the Output panel. You can also check and set the variable’s values in the Debugger in test mode.

Scoping and declaring variables

A variable’s scope refers to the area in which the variable is known and can be referenced.

- Time line variables

Timeline variables are available to any script on that timeline. To declare timeline variables, use the *var* statement and initialise them in any frame in the timeline; the variable will be available to that frame and all following frames, as shown in the following example:

```
//initialised in Frame 1, it is available to all frames
```

```
var x:Number = 15;
```

♦ Local variables

To declare local variables, use the *var* statement inside the body of a function. A local variable declared within a function block is defined within the scope of the function block and expires at the end of the function block.

```
    var firstName:String = "Fred";
```

♦ Global variables

Global variables and functions are visible to every timeline and scope in a project. To create a variable with global scope, use the *_global* identifier before the variable name. For example, the following code creates the global variable *myName*:

```
_global.myName = "George"; // correct syntax
```

Strict data typing

ActionScript 2.0 enables to explicitly declare the object type of a variable when you create it, which is called strict data typing. Strict data typing offers several benefits at compile time.

- Because data type mismatches trigger compiler errors, strict data typing helps you find bugs in your code at compile time and prevents you from assigning the wrong type of data to an existing variable.
- During authoring, strict data typing activates code hinting in the ActionScript editor.
- Strict data typing increases performance at runtime. By indicating the type of data a variable will hold, the Flash player saves a lot of time, in comparison to having to figure out on its own the data type of a variable.
- Using strict typing also helps to ensure that you do not attempt to access properties or methods that are not part of an object’s type.

To assign a specific data type to an item, you specify its type using the *var* keyword, as shown in the following examples:

➔ strict typing of variables

```
    var x:Number = 7;
    var myVariable:String = "Hello";
    var birthday:Date = new Date();
```

➔ strict typing of parameters

```
    function welcome(firstName:String, age:Number){
    }
```

➔ strict typing of parameter and return value

```
    function square(x:Number):Number {
      var squared:Number = x*x;
```

```
        return squared;
      }
```

If you are implementing strict data typing, make sure you are publishing files for ActionScript 2.0.

Manipulating values in expressions

An expression is any statement that Flash can evaluate and that returns a value. You can create an expression by combining operators and values or by calling a function.

Operators are characters that specify how to combine, compare, or modify the values of an expression. The elements that the operator performs on are called *operands*. For example, in the following statement, the addition (+) operator adds the value of a numeric literal to the value of the variable *fido; fido* and *3* are the operands:

```
  fido + 3
```

When two or more operators are used in the same statement, some operators take precedence over others. ActionScript follows a precise hierarchy to determine which operators to execute first. For example, multiplication is always performed before addition; however, items in parentheses ()] take precedence over multiplication. So, without parentheses, ActionScript performs the multiplication in the following example first:

```
  total = 2 + 4 * 3;
```

The result is 14.

But when parentheses surround the addition operation, ActionScript performs the addition first:

```
  total = (2 + 4) * 3;
```

The result is 18.

Comparison operators

Comparison operators compare the values of expressions and return a Boolean value (*true* or *false*). These operators are most commonly used in loops and in conditional statements. In the following example, if the variable *score* is 100, a certain function is called; otherwise, a different function is called:

```
  // call one function or another based on score
  if (score > 100){
    highScore();
  }
  else {
    lowScore();
  }
```

In the following example, if the user's entry (a string variable, *userEntry*) matches their stored password, the play bar moves to a named frame called *welcomeUser*:

```
  if (userEntry == userPassword) {
    gotoAndStop("welcomeUser");
  }
```

Uppercase characters precede lowercase characters in alphabetical order, so “Eagle” comes before “eagle” If you want to compare two strings or characters regardless of case, you need to convert both strings to upper- or lowercase before comparing them.

The following table lists the ActionScript comparison operators:

Operator	Operation performed
<	Less than: Returns true if the left operand is mathematically smaller than the right operand. Returns true if the left operand alphabetically precedes the right operand (for example, a < b).
>	Greater than: Returns true if the left operand is mathematically larger than the right operand. Returns true if the left operand alphabetically follows the right operand (for example, b > a).
<=	Less than or equal to: Returns true if the left operand is mathematically smaller than or the same as the right operand. Returns true if the left operand alphabetically precedes or is the same as the right operand.
>=	Greater than or equal to: Returns true if the left operand is mathematically larger than or the same as the right operand. Returns true if the left operand alphabetically follows or is the same as the right operand.
<> !=	Not equal: Returns true if the operands are not mathematically equivalent. Returns true if the operands are not the same.
== note the two equal signs	Equality: Tests two expressions for equality. The result is true if the expressions are equal.
===	Strict equality: Tests two expressions for equality; the strict equality operator performs the same as the equality operator except that data types are not converted. The result is true if both expressions, including their data types, are equal. Does not apply to strings.

String operators

The addition (+) operator has a special effect when it operates on strings: It concatenates the two string operands. For example, the following statement adds "*Congratulations,*" to "*Donna!*"

"Congratulations, " + "Donna!"

The result is *Congratulations, Donna!* If only one of the addition (+) operator’s operands is a string, Flash converts the other operand to a string. ActionScript treats spaces at the beginning or end of a string as a literal part of the string.

The comparison operators >, >=, <, and <= also have a special effect when operating on strings. These operators compare two strings to determine which is first in alphabetical order. The comparison operators compare strings only if both operands are strings. If only one of the operands is a string, ActionScript converts both operands to numbers and performs a numeric comparison. Uppercase characters precede lowercase in alphabetic order, so “Eagle” comes before “eagle.” If you want to compare two strings or characters regardless of case, you need to convert both strings to upper- or lowercase before comparing them.

Logical operators

Logical operators compare Boolean values (*true* and *false*) and return a third Boolean value. For example, if both operands evaluate to *true*, the logical AND (*&&*) operator returns *true*. If one or both of the operands evaluate to *true*, the logical OR (||) operator returns *true*. Logical operators are often used with comparison operators to determine the condition of an *if* statement. For example, in the following script, if both expressions are true, the *if* statement will execute and the *myFunc()* function will be called:

```
if (i > 10 && i <50){
  myFunc(i);
}
```

The following table lists the ActionScript logical operators:

Operator	Operation performed
&&	Logical AND: Returns true only if both the left and right operands are true.
||	Logical OR: Returns true if either the left or right operand is true. The sign is SHIFT Backspace (\\)
!operand	Logical NOT: Returns the logical (Boolean) opposite of the operand. The logical NOT operator takes one operand.

Bitwise operators

Bitwise operators internally manipulate floating-point numbers to change them into 32-bit integers. The exact operation performed depends on the operator, but all bitwise operations evaluate each binary digit (bit) of the 32-bit integer individually to compute a new value.

The following table lists the ActionScript bitwise operators:

Operator	Operation performed
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Shift left
>>	Shift right
>>>	Shift right zero fill

Equality operators

You can use the equality (==) operator to determine whether the values or references of two operands are equal. This comparison returns a Boolean (*true* or *false*) value. If the operands are strings, numbers, or Boolean values, they are compared by value. If the operands are objects or arrays, they are compared by reference.

➔ It is a common mistake to use the assignment operator to check for equality. For example, the following code compares *x* to 2:

```
if (x == 2)
```

In that same example, the expression

```
if (x = 2)
```

is incorrect, because it does not compare the operands; it assigns the value of 2 to the variable x.

The strict equality (===) operator is similar to the equality operator, with one important difference: The strict equality operator does not perform type conversion. If the two operands are of different types, the strict equality operator returns false. The strict inequality (!==) operator returns the opposite of the strict equality operator.

The following table lists the ActionScript equality operators:

Operator	Operation performed
==	Equality
===	Strict equality
!=	Inequality
!==	Strict inequality

Assignment operator =

You can use the assignment (=) operator to assign a value to a variable, as shown in the following example:

```
var password:String = "Sk8tEr";
```

You can also use the assignment operator to assign multiple variables in the same expression. In the following statement, the value of *d* is assigned to the variables *a*, *b*, and *c*:

```
a = b = c = d;
```

You can also use compound assignment operators to combine operations. Compound operators perform on both operands and then assign the new value to the first operand. For example, the following two statements are equivalent:

```
x += 15;
x = x + 15;
```

The assignment operator can also be used in the middle of an expression, as shown in the following example:

```
// If the flavour is not vanilla, output a message.
if ((flavour = getIceCreamFlavour()) != "vanilla") {
  trace ("Flavour was " + flavour + ", not vanilla.");
}
```

This code is equivalent to the following code, which is slightly easier to read:

```
flavour = getIceCreamFlavour();
if (flavour != "vanilla") {
    trace ("Flavour was " + flavour + ", not vanilla.");
  }
```

The following table lists the ActionScript assignment operators:

Operator	Operation performed
=	Assignment
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
%=	Modulo and assignment
/=	Division and assignment
<<=	Bitwise shift left and assignment
>>=	Bitwise shift right and assignment
>>>=	Shift right zero fill and assignment
^=	Bitwise XOR and assignment
|=	Bitwise OR and assignment
&=	Bitwise AND and assignment

Repeating actions, loops

ActionScript can repeat an action a specified number of times or while a specific condition exists. Use the *while*, *do..while*, *for,* and *for..in* actions to create loops.

```
//a for loop that adds numbers 1-10
var sum:Number = 0;
for (var i=1; i<=10; i++) {
  sum += i;
}
```

Using functions

Flash has built-in functions that let you access certain information and perform certain tasks, such as getting the version number of Flash Player that is hosting the SWF file *getVersion()*.

You can define functions to execute a series of statements on passed values. Your functions can also return values. After a function is defined, it can be called from any timeline, including the timeline of a loaded SWF file.

Functions that belong to an object are called *methods*.

Defining a function

As with variables, functions are attached to the timeline of the movie clip that defines them, and you must use a target path to call them. As with variables, you can use the *_global* identifier to declare a global function that is available to all timelines and scopes without using a target path. To define a global function, precede the function name with the identifier *_global*, as shown in the following example:

```
_global.myFunction = function (x:Number):Number {
   return (x*2)+3;
}
```

To define a timeline function, use the *function* statement followed by the name of the function, any parameters to be passed to the function, and the ActionScript statements that indicate what the function does.

The following example is a function named *areaOfCircle* with the parameter *radius*:

```
function areaOfCircle(radius:Number):Number {
  return Math.PI * radius * radius;
}
```

Passing parameters to a function

Parameters or arguments are the elements on which a function executes its code. For example, the following function takes the parameters/arguments of *initials* and *finalScore*:

```
function fillOutScorecard(initials:String,
finalScore:Number):Void {
  scorecard.display = initials;
  scorecard.score = finalScore;
}
```

When the function is called, the required parameters must be passed to the function. The function substitutes the passed values for the parameters in the function definition. In this example, *scorecard* is the instance name of an object; *display* and *score* are properties of the object. The following function call assigns the value *"JEB"* to the variable *display* and the value 45000 to the variable *score*:

```
fillOutScorecard("Fred", 45000);
```

The parameter *initials* in the function *fillOutScorecard()* exists while the function is called and ceases to exist when the function exits. If you omit parameters during a function call, the omitted parameters are passed as *undefined*. If you provide extra parameters in a function call that are not required by the function declaration, they are ignored.

Returning values from a function

Use the *return* statement to return values from functions. The *return* statement stops the function and replaces it with the value of the *return* statement.

For example, the following function returns the square of the parameter *x* and specifies that the returned value must be a Number:

```
function sqr(x:Number):Number {
  return x * x;
}
```

Calling a user defined function

You can use a target path to call a function in any timeline from any timeline, including from the timeline of a loaded SWF file. If a function was declared using the *_global* identifier, you do not need to use a target path to call it.

To call a function, enter the target path to the name of the function, if necessary, and pass any required parameters inside parentheses. For example, the following statement invokes the function `sqr()` in the movie clip *mathLib* on the main timeline, passes the parameter 3 to it, and stores the result in the variable temp:

```
var temp:Number = this.mathLib.sqr(3);
```

The following example uses a path to call the *initialize()* function that was defined on the main timeline and requires no parameters:

```
this.initialize();
```

The following example uses a relative path to call the *list()* function that was defined in the *functionsClip* movie clip:

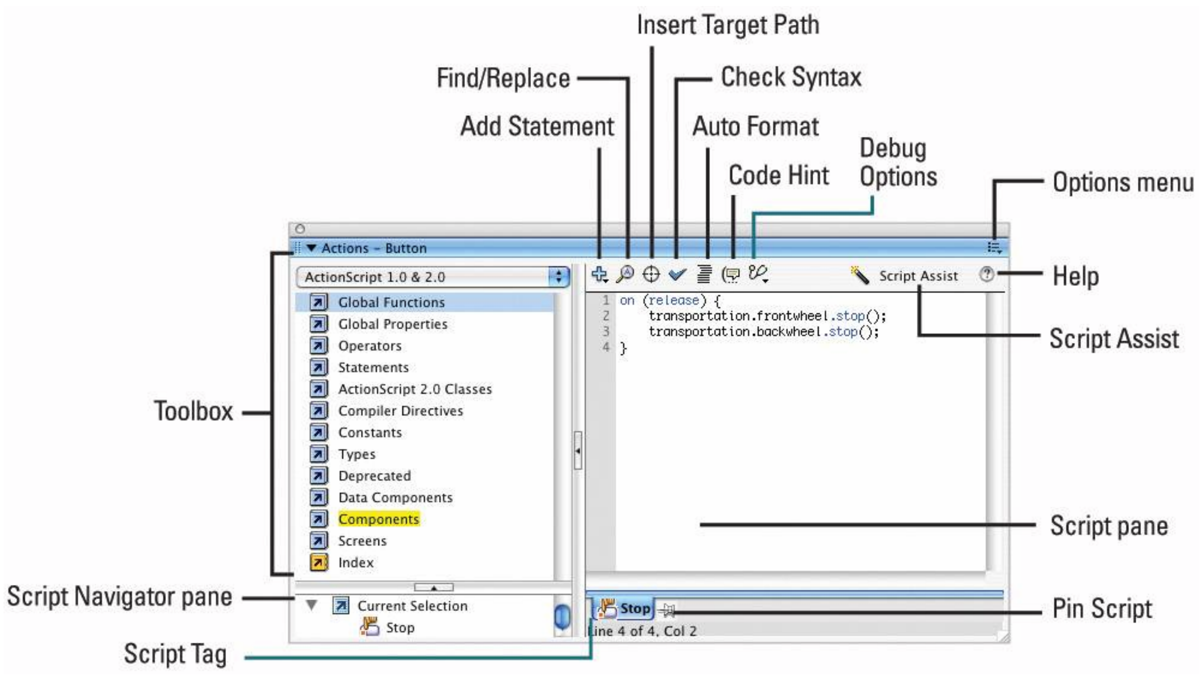
```
this._parent.functionsClip.list(6);
```

Viewing the Actions Panel

The Action panel is where the Flash designer gains control of a Flash document, by allowing you to create and edit actions for an object or frame. To use the Actions panel, first select an object on the stage, or select a frame on the Timeline, then click the Window menu, and then click Actions. Scripts can be typed directly into the Actions panel using the Script pane, or augmented by using a list of installed Actions in the Toolbox.

To open the Action panel, **press F9** or select Window > Development Panels > Actions.

- Toolbox. Supplies a list of all installed actions, organized into a folder.
- Script pane. Enter the Actions into the Script pane.
- Script Navigator pane. Gives reference to all the Scripts in the active movie.
- Current Script tag. Indicates which script is being edited.
- Pin Script. Adds a tab for a selected script.
- Options menu. Contains options that control and format the Actions panel.
- Add Statement. Lets you add script elements to the current action.
- Find, and Find and Replace. Searches the active script.
- Insert Target Path. Inserts a specific target clip into the action.
- Check Syntax. Checks the current action for syntax errors.
- Auto Format. Cleans up the script by auto indenting.
- Show Code Hint. Gives you hints to the syntax of the action, as you type.
- Debug Options. Let's you add or remove breakpoints into the action, which causes Flash to pause on the specified line of code.
- Script Assist. Script assist provides a visual interface for editing scripts that includes automatic syntax completion and parameter descriptions.
- Help. Provides online help.



```
on (release) {
    transportation.frontwheel.stop();
    transportation.backwheel.stop();
}
```

Using the Actions Panel/ActionScript Editor

Where to place ActionScript?

You control a Flash movie by placing actions, that is, instructions or commands written in the ActionScript language, at key places in the movie. You can place actions:

- On **frames**. Actions attached to frames are called frame actions.
  - Frame actions are executed when the play bar reaches the frame that contains the action. Frame actions are executed automatically and do not require input from the user.
- On **buttons**, or **movie clips**. These actions are called object actions.
  - A button action is carried out when the user interacts in the form of a mouse event, such as clicking or rolling over the button.
  - A movie clip action can be carried out in one of two ways, depending on how you define the action: either by an event in the movie, such as a movie clip loading, or by the user clicking the movie clip, which is a mouse event.

Attaching actions to a button called *myButton_btn* looks like the following:

```
on (release) {
  //do something here
}
```

Placing actions with the same purpose in a frame looks like the following:

```
myButton_btn.onRelease = function() {
  //do something here
};
```

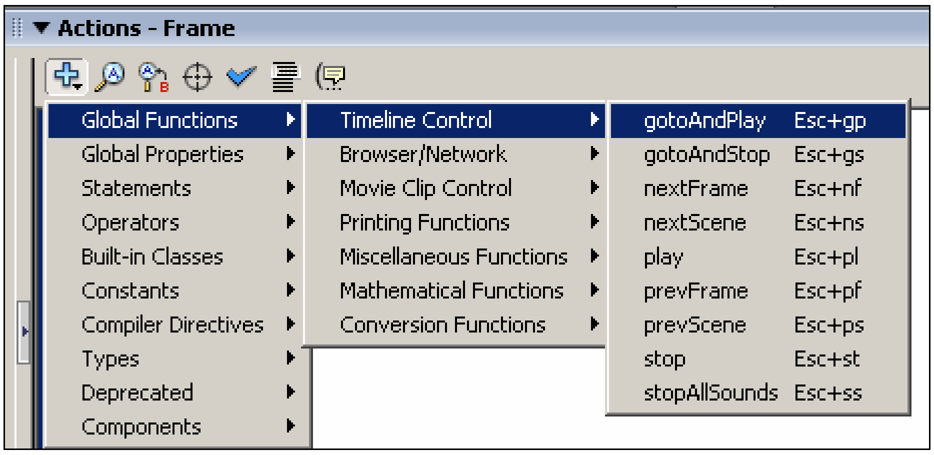
Regardless of where you place the code, the functionality is identical.

The Actions panel

You add actions to Flash movies by using the Actions panel.

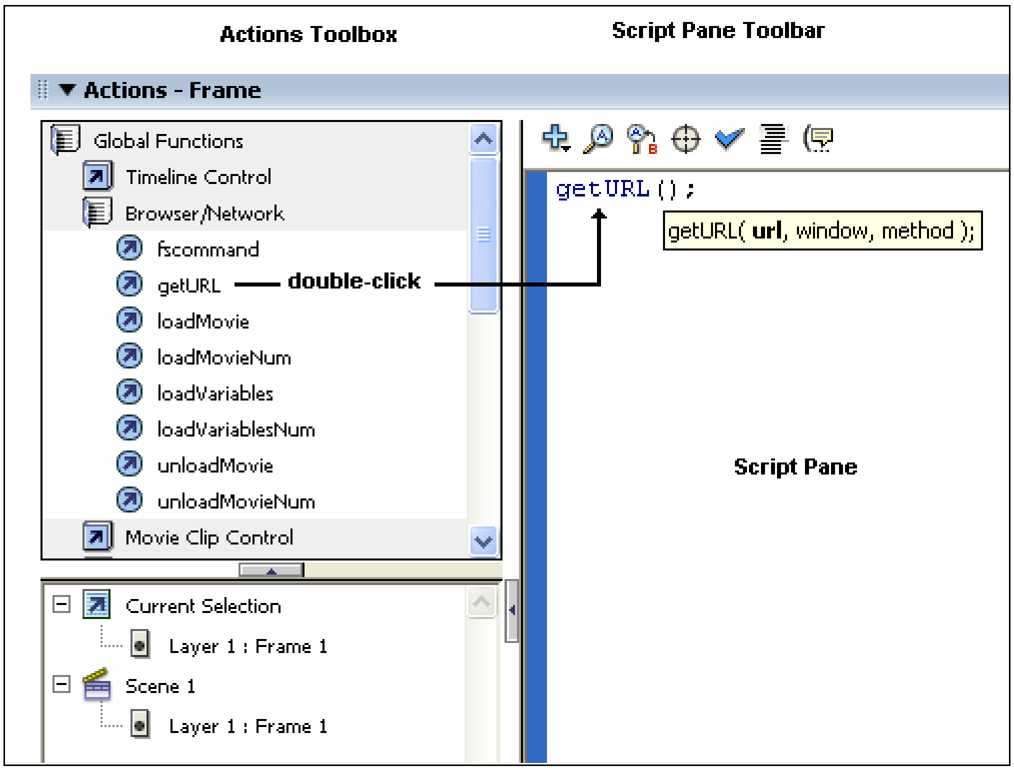
To open the Action panel, **press F9** or select Window > Development Panels > Actions.

- To add an action, you must first select the movie element to which you want to attach the action.
  - A frame. If you select a frame, the Actions panel displays the title Actions – Frame, as shown below.



- A button or movie clip. If you select a button or movie clip, the panel is titled accordingly.

You create ActionScript instructions by clicking the Add (+) button and selecting an action from category submenus.

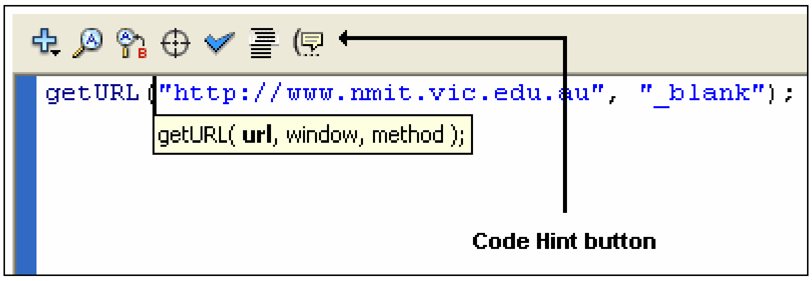


```
getURL();
```
getURL(**url**, window, method);

- The **Actions Toolbox** to the left contains categories of ActionScript elements. Double-clicking an icon expands a category. The Actions toolbox is designed to provide a quick way of adding script elements to your scripts.
- The **Script Pane** is where you add ActionScript. You type into this window just as you would with a word processor.

Many actions require that you enter parameters, which are additional instructions that define how the action works.

- The **Script Pane Toolbar** provides a set of buttons and commands, enabling to add or edit the script in the Script Pane.
  - Once you double-click an icon, or type the opening parenthesis, such as *getURL(*, code hints will appear. You can also manually display code hints at any time by placing the cursor just after the opening parenthesis, then pressing the Code Hint button on the Script Pane Toolbar.



```
getURL("http://www.nmit.vic.edu.au", "_blank");
```

- The **Script Navigator** at the bottom left displays a hierarchical list of elements, such as frames, buttons, movie clips, in your project that contain scripts. Clicking an element will display the script attached to it in the Script Pane for editing purposes.

- References

Flash MX 2004 ActionScript, Peachpit Press ISBN 0-321-21343-2
Flash MX ActionScript advanced, PeachPit Press ISBN 0-201-77022-9
Flash MX ActionScript Bible, Wiley, ISBN 0-7645-3614-1
Flash 5 ActionScript, QUE ISBN 0-7897-2524-X