Written by Billy D. Spelchan for www.BlazingGames.com

Chapter 7

Advanced Action Script

Contents

Now that we have the first game out of the way, we focus on some of the more complex Action Script concepts. While I was tempted to place this chapter in the first part, I felt that it would be a little overwhelming right at the beginning. Instead, I decided to place it right after the first game has been completed, giving the reader time to comprehend the Action Script that was introduced earlier.

- The Switch Statement - A variation of the if then else statement.
- Arrays - Holding multiple variables with one variable.
- While and Do - Basic looping.
- For loop - Looping with a counter.
- It's time for a Break - The break and continue statements.
- Boolean Arithmetic - Logical AND, OR, XOR, NOT and bit shifting.
- Debugging: Output - Trace and the output window.
- Debugging: Breakpoints - Using breakpoints.
- Debugging: Watching variables - watching and changing variables.

The Switch Statement

The Switch statement is a convenient way of handling situations where you have to handle a large number of actions based on the value of a variable. The Switch statement starts of with the switch statement which contains an expression that should evaluate into a number. After the switch statement is a block of code which consists of case statements and an optional default statement.

```
switch (number)
{
      case 1:
            trace("The number is one");
            break;
      case 2:
            trace("The number is two");
            break;
      case 3:
            trace("The number is three");
            break;
      default:
            trace("The number is not one, two or three");

}
```

After each of the case statements is the code you want to execute if the variable being switched equals the value of the case. The code should end with a break statement, which causes the program to skip over the rest of the switch block. If there is no break statement, all code in the following case statement will also be executed. In some cases, this is exactly what you want to happen, as it allows you to define a group of numbers that all do the same action. Forgetting the break statement is a very common mistake for beginners to make.

The default statement will be executed if none of the case statements match the value of the switch expression. No break statement is needed after the default code, though I usually do have a break statement out of habit. While the default statement is optional, it is a good habit to always have a default action, even if the default is simply a trace statement telling you that you have reached code that should not be reached.

Switch statements are a convenience statement designed to replace large numbers of if then else statements. The switch statement we have above could be done with if statements as you can see below. While this works, it is harder to read than a switch block and is slightly less convenient to write.

```
if (number == 1)
{
    trace("The number is one");
}
else if (number == 2)
{
    trace("The number is two");
}
else if (number == 3)
{
    trace("The number is three");
}
else
{
    trace("The number is not one, two or three");
}
```

Arrays

Simply stated, an array is a list. The size of the list determines how many pieces of information, known as elements, can be put into it. Any part of the list can be accessed, as long as you know where in the list it is stored.

Flash has three ways of creating an instance of an array, as follows:

```
x = new Array();
x = new Array(size);
x = new Array (element1, element2, ..., elementN);
```

The first method creates an array that has no elements. While this may not seem useful, it is possible to add new elements to an array, which we will discuss later. The second method creates an array with a specific number of elements. These elements are initially empty, but new values can be assigned to them, as we will discuss later. The final method creates a populated array. The size of the array will depend on the number of elements that are declared, but in theory any number of elements can be specified. This type of constructor is very useful in cases when you want to have some type of table of predefined values that can be accessed.

The power of an array comes from the ability to access and modify the elements of the array. Elements of an array have an index number associated with them. The first element of an array is referenced with the number 0, the second is 1, and so on. Some programmers find element 0 confusing, so will simply ignore that element and start storing stuff in element 1. If you are going to do this, remember that the last element of an array is one less than the size of the array.

To actually access the element, you simply use the array variable's name, followed by the index number in square brackets. The index does not have to be a number, but can be a variable that contains the index number. For example, the following program adds the third and fourth elements (index values 2 and 3) and places the result in the first (index 0) element.

```
index = 3;
list = new Array(0, 1, 2, 3, 4, 5);
result = list[2] + list[index];
list[0] = result;
```

The Array class has functions for modifying and getting information about the array. It also has a read-only property variable named length that holds the current length of the array. Action Script allows you to change the size of the array at any time. Many other programming languages are not quite as flexible. The Action Script Dictionary that is included with Macromedia Flash covers these functions in detail.

While and Do

Looping is a very common programming task. A loop is essentially a block of code that executes until certain conditions are met. Flash supports three types of looping statements. The while statement, the do ... while statement, and the for statement. The first two types of loops are very similar so we will look at them together.

The While statement always has a statement or block of statements tied to it. The condition of the while statement is tested. If the condition is true then the statement or block of code gets executed. When the block of code is finished executing the while statement condition is tested again. If it is still true, the statement or block gets executed again. This continues until the condition is false, at which point the program continues execution on the line after the statement of block of statements. Here is a use of the while statement to see how long an object will fall before hitting the ground.

```
var distRemain = meters;
var vel = 0;
var time = 0;

 while (distRemain > 0)
{
      vel += 9.8;
      distRemain -= vel;
      ++time;
}
```

One thing that starting programmers have to watch out for is the infinite loop. This is a loop that will never end, meaning that the program is stuck until stopped by the user. Flash will only let loops run a certain amount of time before brining up a dialog box telling the user about the problem and letting the user cancel the script.

It is possible that the statement or block after the while statement will never be executed. Sometimes you always want the statement or block to be executed at least once. This is what the do ... while statement does. In this example we want to find the length of time an object will decay rounded up to the nearest half life interval. A half life is the length of time a decaying object takes to be reduced to half of it's original mass.

```
var life = startinglife;
var duration = 0;
do
{
      life /= 2;
      ++duration;
}
while (life > 1);
trace ("Lifespan of object is " + (duration * halflife));
```

For loop

The most common type of loop in programming is the for loop. The purpose of this type of loop is to count through a series of numbers. The format of the for statement is:

```
for (start_condition; end_condition; increment){ /*loop action*/ }
```

The start condition starts the variable that will be counting to it's initial value. This will generally be 0 or 1, but can actually be any value. If there is no need to initialize a variable, you can skip this part of the statement by simply having the semicolon. I would consider having a /*none*/ comment before the semi-colon so you know that there is no loop initialization.

The end condition is simply a boolean statement like you would use with an if statement. The loop continues until this condition is no longer true. The condition can be any boolean condition, and could even consist of a function call.

The increment portion is where the counting variable is changed. It is called at the end of every loop iteration. Generally you will be increasing the counter by 1, but it is valid to decrease the counter, or do any other mathematical operation to the counter.

The following sample will add the numbers 1 to 10 together.

```
var cntr, value = 0;
for (cntr = 1; cntr <= 10; ++cntr)
{
      value += cntr;
}
```

The for statement seems complex. When you get right down to it, the for statement is really a macro statement that makes a while loop behave as a counter. One way of better understanding a for loop is to look at the loop as a while loop. By doing this, you will see what the three parts inside the for loop really do. Here is the above example as a while loop.

```
var cntr, value = 0;
cntr = 1; // the start condition
while(cntr <= 10) // the end condition
{
    value += cntr; // the loop action
    ++cntr; // the increment
}
```

Now we come to a rather interesting additional for statement that is included with Action Script, the for..in statement. This is what is known as an iterative for function. It is used to iterate through all the variables of the indicated object. The format for this command is

```
for (variable_holding_index in object)
```

The variable_holding_index is simply the name of the variable that holds the index. Here is a sample to illustrate how this can be used.

```
var cntr;
testArray = new Array("A", "B", "C", "D");

for (cntr in testArray)
{
      trace("Element " + cntr + " is " + testArray[cntr]);
}
```

Now, notice that the output for this test is the following.

```
Element 3 is D
Element 2 is C
Element 1 is B
Element 0 is A
```

It's time for a Break

Sometimes you may not want a loop to finish looping. For instance, lets say you were searching an array for the first occurrence of a particular number. Once the number has been found, there really is no use in continuing going through the loop. That is why the break statement exists.

While you have seen break used in the switch statement, it can also be used in all types of loops. When it is encountered in a loop, execution of the loop automatically ends and the program continues executing on the first statement after the loops statement or block of statements.

```
for (cntr = 0; cntr < someArray.length; ++cntr)
{
      if (someArray[cntr] == desiredValue)
      {
            targetIndex = cntr;
            break;
      }
}
```

In addition to the break statement, there is also a continue statement. This statement is similar to the break statement but instead of terminating the loop, it simply skips over the rest of the code block and goes immediately to the condition statement. Obviously, if the condition is now false, the loop ends. If the condition is true, however, another iteration of the loop begins. The following example uses the continue to help count only the odd number that appear in an array of numbers. While there are much better ways of accomplishing this task, it does demonstrate the continue statement.

```
for (cntr = 0; cntr < someArray.length; ++cntr)
{
      if ((someArray[cntr] % 2) == 0) // if even
      {
            continue;
      }
      ++oddCount;
      oddSum += someArray[cntr];
}
```

While taking the modulus of the number divided by two is one way to check if a number is even, a much better way is to use boolean arithmetic. We will look at boolean arithmetic next.

Boolean Arithmetic

Boolean arithmetic is based on base 2 – also known as binary – which is the number system that computers use. Humans use the base 10 – better known as decimal – system. While understanding binary is not required to program computers, that knowledge can come in very handy. It can also boost the speed of your program.

Binary numbers work the same way as decimal numbers. Every digit in the number exponentially increases the value of that position. In decimal, 10 is worth ten ones. 100 is worth ten tens. One thousand is worth ten hundreds. In binary 10 is worth two ones (a decimal value of 2). Likewise, 100 is worth two 10's for a decimal value of 4 and 1000 is worth two 100's for a decimal value of 8. Here is a table of the first 16 binary positions. I am counting from 1, though some books consider the first bit position to be 0.

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Value	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

In the decimal system the number 742 can be considered to be 7 x 100 + 4 x 10 + 2. Binary works the same way so the number 1011 could be considered 1 x 1000 + 1 x 10 + 1. To convert to decimal we simply need to add the values of the individual place numbers to get 8 + 2 + 1 = 11. Converting from decimal to binary, we have a bit harder problem. The technique I learned in math class was the division method. With this method, you start with the decimal number and divide that number by 2. If there is a remainder, you place a 1 in as the rightmost spot, otherwise you place a 0. You take the number you got from the division and divide again, placing the remainder in the next position working right to left. This continues until you end up with 1, which goes to the far left position of the resulting binary number. Here is an example:

10 / 2 = 5 r 0 so our binary number is 0.
5 / 2 = 2 r 1 so our binary number is 10.
2 / 1 is 1 r 0 so our binary number is 010.
1 is 1 so we are done with the final resulting number being 1010.

Because binary numbers tend to get very long quickly, programmers started using octal (base 8) and eventually hexadecimal (base 16). These were used because 8 holds 3 bits while 16 holds 4 bits, making numbers much easier to deal with. Hexadecimal is like decimal, but instead of 10 you use the symbol A, 11 is B, 12 is C, 13 is D, 14 is E and 15 is F. Converting from binary to hexadecimal and back can be done in groups of 4 which is known as a nibble. A byte is 8 bits, so a byte can be represented as a two digit hexadecimal number.

Now we are ready to look at some binary specific operations, better known as boolean operations. These operations are designed to work on integers. Considering that Action Script is a typeless language, this is a very interesting distinction. Flash handles this problem by automatically converting the variables used into 32 bit integers before doing the operations. The operations we are going to briefly look at are logical AND, logical OR, logical XOR (exclusive or), bitwise left shift, bitwise right shift and logical NOT.

The first three of these operations work on individual bits. As the numbers being operated on are 32 bits, each of the 32 bits in the first variable is teamed up with the bit in the same position of the second number.

AND (which uses the & symbol)looks to see that both bits are 1. If they both are one the resulting bit is a 1 otherwise the resulting bit is a 0. OR (uses the | symbol) looks to see if either or both bits are 1. If one of the two bits or both of the two bits are 1 the resulting bit is 1 otherwise the resulting bit is 0. XOR (uses the ^ symbol), which stands for exclusive or checks to see if one of the two bits, and only one of the two bits, are 1. If only one of the two bits is set, the result is 1 otherwise the result is 0. While this may not seem useful, XOR is used to toggle specific bits making it very nice. The following table summarizes these operations.

First Bit	Second Bit	AND result	OR result	XOR result
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise shifting is simply the act of taking the bits that make up the integer and moving them one position to the left or to the right. For left shifting, the leftmost bit of the original number is lost and the rightmost bit becomes 0. For right shifting, the rightmost bit of the original number is lost and the leftmost bit becomes 0. The format is

```
variable << number_of_positions_to_shift.
variable >> number_of_positions_to_shift.
```

The final boolean operation, logical NOT (the ~ symbol), simply inverts all the bits in the number.

Debugging: Output

As we are going to be writing a lot of Action Script it would be a good idea to talk about the subject of debugging. There is a lot to cover, so I will break the topic down into three separate sections. The first section, output, looks at the trace statement and the output window. The second section, breakpoints, takes a look at this now-vital method of debugging. Finally, the watch variables section covers watching and changing variables.

In the early days of programming, there really were not debuggers. Instead programmers had to rely on the output their programs generated. As the output was text, this was not that bad as programmers could simply add print statements to print out information about where they were in the program and the state of key variables.

This technique can still be used today. Flash actually has a special output console that it can send text to. To write to this console you simply use the trace statement as follows.

```
trace("I am here!");
trace("The player has " + points +" points");
```

In addition to being a substitute for the debugger, tracing can prove quite useful as you can use it to build an event log for your program. If you are trying to develop a complex game this can be useful as flash can log everything that is happening in the game. Even better, the final movie can be generated without trace statements so you don't even have to remove them for your final product.

In addition to the trace statement, the output window has some really useful features. These features are available even if you are not running the debugger! The debug menu has two items that send output to the output window. They are "list variables" and "list objects".

List variables sends a list of all the program's variables to the output window. If your program suddenly hangs or is doing something really weird and you are not running the debugger (I tend to only use the debugger as a last resort, or when I am doing extended testing and need to be able to change variables manually) you can still dump the variables to the output window and hopefully find out what is going wrong without having to restart in the debugger.

List objects is like listing the variables but instead of showing a list of variables it shows all the objects that are currently in use on the current frame.

The ability to dump variables and objects even when you are not debugging a program is certainly useful. That, however, is not where the power of the output window ends. You can copy the output window to the clipboard or send the contents of the window to the printer. You are also able to save the contents of the output window to a file. You can even search the output window

for an occurrence of a string of text!

Debugging: Breakpoints

Flash makes developing movies which contain a lot of Action Script easier by having a debugger built into Flash. The debugger is a tool which helps you find bugs within your movie. Perhaps the most important thing you can do with a debugger is set break points and step through your program. A breakpoint is essentially a marker. When the program reaches the marker, it pauses. While paused, you have the ability to move through the program line by line.

Setting a breakpoint is very easy. All you have to do is move the cursor to the line you want to place a breakpoint on. You can then right click on the line and select "set breakpoint" from the menu. Breakpoints can also be set and modified from the debugger.

To use the breakpoints, run the movie by using the "Debug the Movie" command. When your program reaches one of the breakpoints that you set it will pause. To continue the movie, click on the play button. Alternatively you can step through the movie by using the step over or the step through commands.

Both functions let you step line by line through the program. The big difference between the two commands is what they do when the code that is about to execute is a function. Step over will run the function as if the function was a normal statement. In other words, The function runs, but you are not shown the code of the function. This is usefull for cases where you know the function about to be run is bug free.

Step through actually takes you inside the function where you can step line by line in the function to see what is going on. If you think your problem might be within the function that is being called then step through is ideal.

Another good use of step through function is when you are going through someone elses code. Stepping through the code gives you a much better idea of what is going on.

Debugging: Watching variables

Knowing what code your Action Script is running is useful, but to take even more advantage of this knowledge, knowing the contents of the variables is extremely useful. The debugger automatically shows you the values of every element in the movie. This list of variables is handled the same way as the movie browser component. In other words, this is a branching tree containing the objects within this movie. The global value, being what is globally defined. Level0 is the actual movie that we are running. All the movies that are part of this movie are listed here.

While this is all fascinating, what you want to do is to be able to see the variables that you are dealing with. These are visible by simply selecting _level0. You then have to select the variables tab from the list of tabs. Then you will notice a long list of variables (some of which are names of functions, which is why you can assign functions to variables). This is fine, but most likely you will only be interested in a few variables, so having all the variables gets in the way of things.

This is where the watch tab comes into play. If you go to the watch tab you will notice that it is empty. How is an empty tab useful? It isn't. To make it useful we need to add items to this tab. Go back to the variables tab. Now right click on the variable that you want to keep track of. The watch option becomes available. Select it to enable watch. A dot will appear by the name of the variable to remind you that you are watching this variable. Now, if you return to the watch tab, you will notice that the variable is listed there! You can watch as many variables as you desire. To stop watching the variable, just right click it again and select the now-checked watch item.

The key to effectively debugging is to only watch variables that are of interest to you for the task that you are debugging. You can always use the variable tab to find information about other variables, so only variables that should affect your problem should be watched. If you watch too many variables, you end up wasting time.

Another thing to remember about the watch window is that the variables it shows there are the watched variables that are currently in scope. In English, this means that only the variables that are currently visible to your program are shown in the watch window. If you watch a variable that is declared using the var statement, it will only be visible when you are inside the function that defined it.

In fact, forgetting to define a variable using var causes that variable to be visible to the entire movie it is defined in. This can sometimes lead to hard to find bugs, but by carefully scrutinizing the variable tab you can find instances of when this is the case.

The variable window lets you do more than look at the value of variables, it also allows you to change the value of variables. All you need to do is type in a new value for the variable.You are also able to change the properties of objects in the movie through the properties tab. Not all properties are modifiable, however.

Why is changing the value of variables while debugging useful? I have found few cases where it is useful for debugging. For general testing of a program, however, it is a really nice feature.

For instance, let us say you wanted to test a movie with different values. You could use the time consuming method of running through the game multiple times making changes in your code to change the value of the movie. This works, but if you forget to change the values back to their original it could lead to problems later. By setting a break point just after the variables have been set you can then modify the variables and put the values you want in there. As no code is changed, you don't have to worry about code changes.