CS 116 – Action Script Expressions, Statements & Operators

Elie Abi Chahine

Operators

- What are operators?
  - Exactly like in math, operators are used with numbers and variables (aka operands) to create expressions.
  - Eg: x = y + 17 in here we used 2 operators, the "=" and the "+"

Operators

- Another example?

```
var uiVariable:uint = 2 + 3 * 4;     /* uiVariable = 14 */
```

In the code above, the addition **(+)** and multiplication **(*)** operators are used with three literal operands **(2, 3, and 4)** to return a value. This value is then used by the assignment **(=)** operator to assign the returned value, **14**, to the variable **uiVariable**.

Operators

- Operators can be unary, binary, or ternary.

  – A unary operator takes one operand. For example, the increment **(++)** operator is a unary operator, because it takes only one operand.

```
Ex: varnBlah:Number=0;    /* variable Blah is equal to 0 */
    nBlah++;              /* Blah is equal to 1 */
```

  – A binary operator takes two operands. For example, the division **(/)** operator takes two operands.

```
Ex:     nBlah = 10/2;     /* 10 and 2 are the operands */
```

  – A ternary operator takes three operands. For example, the conditional **(?:)** operator takes three operands.

```
Ex:     nBlah==5 ? nBlah=0 : nBlah=2;
```

Operators

- Some operators are overloaded, which means that they behave differently depending on the type or quantity of operands passed to them.

- The addition (+) operator is an example of an overloaded operator that behaves differently depending on the data type of the operands..

```
Ex:   trace(5 + 5);          /* 10 If both operands are numbers, the addition
                                operator returns the sum of the values */

      trace("5" + "5");   /* 55 If both operands are strings, the addition
                             operator returns the concatenation of the two
                             operands. */
```

Precedence & Associativity

- Let's start with examples:

5 + 3 * 8 is it: 5 + 3 * 8 = 8 * 8 = 64
or: 5 + 3 * 8 = 5 + 24 = 29

Of course it is the second one, which gives us the rule of precedence. Some operators have higher priority (precedence) which makes the compiler run them first in an expression.

8 - 5 - 1 is it: 8 - 5 - 1 = 3 - 1 = 2
or: 8 - 5 - 1 = 8 - 4 = 4

It can be tricky. Well the answer is 8-5-1=2 which means that the first "–" operator was done first by the compiler.

All this to show you the associativity rules. Some operators are left-associative and others are right-associative. In our case, the minus operator is left associative.

Precedence

- In the next slide you will find the table listing the operators for ActionScript 3.0 in order of decreasing precedence.

- Each row of the table contains operators of the same precedence. Each row of operators has higher precedence than the row appearing below it in the table.

Precedence Table

Group	Operators
Primary	[ ] {x:y} () f(x) new x.y x[y] <> </> @ :: ..
Postfix	x++ x--
Unary	++x --x + - ~ ! delete typeof void
Multiplicative	* / %
Additive	+ -
Bitwiseshift	<< >> >>>
Relational	< > <= >= as in instanceof is
Equality	== != === !==
BitwiseAND	&
BitwiseXOR	^
BitwiseOR	|
LogicalAND	&&
LogicalOR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= >>>= &= ^= |=
Comma	,

Associativity

- You may encounter situations in which two or more operators of the same precedence appear in the same expression.

- In these cases, the compiler uses the rules of associativity to determine which operator to process first.

- All of the binary operators, except the assignment operators, are left-associative, which means that operators on the left are processed before operators on the right.

- The assignment operators(= , += , -= , ect…) and the conditional (?:) operator are right-associative, which means that the operators on the right are processed before operators on the left.

Associativity

- Ex:

```
trace(3 > 2 > 1);              /* false   but WHY???   */
```

- How the compiler sees it:

```
trace(3 > 2 > 1);

  trace(true > 1);   /* true is equal to 1 for the compiler */

     trace(1 > 1);     /* is 1 greater than 1???   */

        trace(false);     /* false is 0 for the compiler */
```

Output: false

Operators

- I am going to cover the mostly used operators in this chapter with some examples.

- If you are curious to know what all operators do check the “flash_actionscript3_programming” pdf (found on moodle) page 112 to 116.

Note: Later in the semester, I will be covering other operators depending on the topic we will be covering.

[ ] () f(x) x.y x[y]

Op	Meaning	Example
[ ]	Initializes an array.	var a:Array = [1, 2, 3, 4]
( )	Groups expressions	8 – (5 – 1) = 8 – 4 = 4

<u>Note:</u> ***Even if the "-" operator is left-associative the () operator gave priority to the second "-" sign***

Op	Meaning	Example
f(x)	Calls a function	trace("This is starting to make sense, NOT")
x.y x[y]	Accesses a property	Math.sqrt(25); (the "." is the operator) a[2] = 15; (a being an array)

++ --

Op	Meaning	Example
++	Increments (postfix)	i++;
--	Decrements (postfix)	i--;

Note: in both examples i can be of type Number, int or uint

```
var i:int = 5;
i++;          /* same as i = i+1 */
trace(i);     /* output will be 6 */
i--;          /* same as i = i - 1; */
trace(i);     /* output will be 5 */
```

++ -- + - !

Op	Meaning	Example	
++	Increments (prefix)	++i;	/* i = i + 1 */
--	Decrements (prefix)	--i;	/* i = i - 1 */
+	Unary +	x = +5;	/* not used a lot */
-	Unary - (negation)	x = -5;	
!	Logical NOT	b = !b;	/* true becomes false & vise versa */

Note: **what if we apply the ! to an int, uint or Number types?**

```
var i:int = -5;
i = !i;     /* Any number other than 0 becomes a 0 */
trace(i);   /* output will be 0 */
i = !i;     /* i becomes 1 */
trace(i); /* output will be 1 */
```

Prefix vs Postfix (++   --)

The Best way to explain this is with an example:

```
var i:int = 5;
trace(i++); /* output will be 5 */
trace(i);  /* output will be 6 */

var i:int = 5;
trace(++i); /* output will be 6 */
```

Basically, *trace(i++);* is transformed as follows:

```
    trace(i);
    i = i+1;
```

But, *trace(++i);* will be:

```
    i = i+1;
    trace(i);
```

* / % + -

Op	Meaning	Example
*	Multiplication	x = y * 3;
/	Division	x = y / 2;
%	Modulo	x = 5 % 2; /* x will be the remainder of 5/2 which is 1 */
+	Addition	x = y + 2;
-	Subtraction	x = y – 2;

< > <= >= == !=

Op	Meaning	Example
<	Less than	if(x < 10)
>	Greater than	if(x > 10)
<=	Less than or equal to	if(x <= 10)
>=	Greater than or equal to	if(x >= 10)
==	Equality	if(x == 10)
!=	Inequality	if(x != 10)

Note: The relational and equality operators take two operands, compare their values, and return a Boolean value.

```
var x:int = 5;
if( x <= 5 ) /* this will return true */
{
    trace(“ the condition is satisfied”);
}
```

&& ||

Op	Meaning	Example
&&	Logical AND	if(x < 10 && y > 20)
||	Logical OR	if(x < 10 || y > 20)

Note: The && operator needs both conditions to be satisfied so that a true is returned. As for the || operator, one condition satisfied is enough to return a true value.

Cond1	Cond2	&&	||
true	*true*	*true*	*true*
true	*false*	*false*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*

= *= /= %= += -=

Op	Meaning	Example
=	Assignment	x = y + 3;
*=	Multiplication assignment	x *= 3; /* Same as x = x * 3 */
/=	Division assignment	x /= 5; /* Same as x = x / 5 */
%=	Modulo assignment	x %= 7; /* Same as x = x % 7 */
+=	Addition assignment	x += 9; /* Same as x = x + 9 */
-=	Subtraction assignment	x -= 2; /* Same as x = x - 2 */

```
Eg:  var x:int = 5;
     x *= 3      /* this will change the value of x to 15 since x = x * 3 */
     trace(“ the value of x is ” + x );
```

Expressions & Statements

- An expression consists of one or more operand with an operator.
  - **Eg: x = x * speed**

- Once an expression ends with a semicolon then it becomes a statement.
  - **Eg: x = x * speed;**

```
while ( x <= 10 ) /* this is an expression */
{
  trace (“In the loop ”); /* this is a statement */
  x++; /* this is a statement */
}
```

Expressions & Statements

Guides to writing good code:

- Write expressions and statements in a way that makes their meaning as transparent as possible. That is, write the clearest code that does the job.
- Format to help readability. For example, use spaces around operators to suggest grouping and precedence.
- Indent to show structure.
- An example of a badly formatted is:

```
for(n++;n<100;field[n++]=0);
i = 0; return ('\n’);
```

Expressions & Statements

Guides to writing good code:

- Reformatting improves it a little bit:

```
for(n++; n < 100; field[n++] = 0);
i = 0;
return ('\n’);
```

Expressions & Statements

Guides to writing good code:

- Even better is to put the assignment in the body and separate the increment, so the loop takes a more conventional form and is thus easier to understand for both the reader and the programmer. A huge plus would be adding those brackets to specify the for-loop’s scope:

```
for (n++; n < 100; n++)
{
        field[n] = 0;
}

i = 0;
return ('\n’);
```

- Initially:

```
for(n++;n<100;field[n++]=0);
i = 0; return ('\n’);
```

Expressions & Statements

Guides to writing good code:

- Parenthesize to resolve ambiguity.
- Parentheses specify grouping and can be used to make the programmer’s intent clear even when they are not required.
- ActionScript has lots of nasty precedence problems and it is easy, even for a seasoned programmer, to make a mistake.

```
leapYear = year % 4 == 0 && year % 100 != 0 || y % 400 == 0;
```

- but the use of parentheses makes it easier to understand the structure of the expression:

```
leapYear = (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0);
```

The End ☺