DigiPen
INSTITUTE OF TECHNOLOGY

CS116 - Action Script Fundamentals

Elie Abi Chahine

What's a computer program?

- First of all, it's useful to have a conceptual idea what a computer program is and what it does.
- There are two aspects of a computer program:
  - It is a series of instructions or steps for the computer to carry out. Each individual instruction is known as a statement.
  - Each step ultimately involves manipulating some piece of information or data.

What's a computer program?

- In a simple case, you might instruct the computer to add two numbers and store the result in its memory.
- In a more complex case, imagine there is a rectangle drawn on the screen, and you want to write a program to move it somewhere else on the screen. The computer is keeping track of certain information about the rectangle (the x, y coordinates where it's located, how wide and tall it is, what color it is, and so forth). Each of those bits of information is stored somewhere in the computer's memory.
- A program to move the rectangle to a different location would have steps like "change the x coordinate to 200; change the y coordinate to 150" (in other words, specifying new values to be used for the x and y coordinates). Of course, the computer does something with this data to actually turn those numbers into the image that appears on the computer screen; but for the level of detail we're interested in, it's enough to know that the process of "moving a rectangle on the screen" really just involves changing bits of data in the computer's memory.

What is a programming language?

- A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer.
- Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.
- Many programming languages have some form of written specification of their syntax (form) and semantics (meaning).

What is ActionScript?

- ActionScript is the language you use to add interactivity to Flash applications, whether your applications are simple or complex.

- You don't have to use ActionScript to use Flash, but if you want to provide basic or complex user interactivity, work with objects other than those built into Flash (such as buttons and movie clips), or otherwise turn your application into a more robust user experience, you'll probably want to use ActionScript.

Create a project

- Open Adobe Flash CS5.5 Professional.
- Create a new Flash(ActionScript 3.0) Document.
- Press F9 to get the Actions Tab

Run a project

- Now, inside the actions window, we will write our own code.

- After writing the code, choose from the menu: “Control” then “Test Movie” or hit Ctrl+Enter to run the program.

Example 1: Displaying a message

- In order to display a message in the output window we use the trace function

trace("Welcome to CS116")

- Welcome to CS116 is displayed in the output window. Notice that the string is enclosed between double quotation marks but the output displayed won't have them.

Example 1: Displaying a message (cont'd)

- Now let's try the following code:

```
trace("Welcome to CS116");
trace("I want to get an 'A' in this class");
trace("Welcome to CS116\nI want to get an 'A' in this class");
```

- Every time we call a trace function we jump into a new line
- The same two lines are displayed twice although we only used 3 trace calls
- "\n" in the 3rd trace call was responsible for jumping to a new line
- A similar thing will be using "\t" which will display a tab in the output

- **What about the semicolon at the end of each function call?**
  - Unlike C and C++, the semicolon in ActionScript is not necessary unless you want to separate 2 statements on the same line.
  - trace("Welcome to CS116");trace("I want to get an 'A' in this class")

Example 2: Comments

- Write the following:

```
/* using the trace function */
trace("Welcome to CS116"); // first line
trace("I want to get an 'A' in this class"); /* second line */

/*
Displaying two lines with one trace call
The output should be the same as the two lines displayed
above
*/
trace("Welcome to CS116\nI want to get an 'A' in this class");
```

Example 2: Comments *(cont'd)*

- Comments are used to clarify and explain the code.
- Comments start with a /* and end with a */
- One line comments can also start with //
- The compiler ignores all the comments. In other words, the comments will not be present in the executable code. Therefore, comments have no effect on the program.
- The comments could be placed partially on a line, on an entire line, or on more than one line.

Example 2: Comments *(cont'd)*

Guides to writing good code:

- In CS116, every line of code you write should be commented.
- It is highly recommended to always use the /**/ format even if it is a one line comment.
- Points will be removed in the assignments if no comments are found or if the comments are not helpful for the user to understand the code.

Example 3: Multiple Instruction Program

- Write the following:

```
/* A function that displays the class schedule */
function DisplayClassSchedule()
{
  trace("We have only one section");
  trace("It is 2 times per week");
  trace("Monday and Wednesday");
}

/* A function that displays a welcome phrase */
function DisplayWelcome()
{
  trace("Welcome to CS116");
}

/* Calling the Welcome function */
DisplayWelcome();
/* Calling the ClassSchedule function */
DisplayClassSchedule();
```

Example 3: Multiple Instruction Program *(cont'd)*

- This code contains 2 functions
- To create a function you start with the keyword function followed by the function's name and parentheses.
- A function is made of a set of instruction (can be also made of one instruction)
- A function can be called more than once (actually that is it's main purpose)
- All instructions inside a function will be executed when calling the function
- The function's order of creation doesn't matter, the function calling order is what matters for the execution

Example 3: Multiple Instruction Program *(cont'd)*

Guides to writing good code:

- Function names should be based on active verbs, perhaps followed by nouns.

function DisplayClassSchedule()

- I expect from a function called **GetTime** to actually return the time, so without looking at the code, I should know that this function returns something to the use (a Number, a Date Class…)

- Every function should have a **header**.

- A header is one or more lines of comments put above the function in order to tell the user what that function does and what to expect when calling this function (sometimes functions will change values or settings that the user should be warned about…).

More on this topic will be covered in the "functions" chapter

Example 4: Functions with arguments and returning a value

- Write the following:

```
/* A function that adds two numbers and returns
   the result */
function Add(n1_:Number , n2_:Number):Number
{
    return n1_ + n2_;
}


/* Calling the Add function  and displaying the
   result */
trace("The result is: " + Add(5,6));
```

Example 4: Functions with arguments and returning a value *(cont'd)*

- Function Add specifies that it takes two Numbers and returns a Number
- When a function has more than one argument, a comma is used to separate the arguments.
- The Add function's body is Adding the two numbers sent by parameter and returning the result of that addition
- When calling the function Add(5,6) we are specifying that the variable n1_ will be equal to 5 and n2_ to 6
- Notice that the trace function contains a "+" that is separating a string and a function call. Hence, the string will be displayed followed by the number returned by the Add function.

Example 5: Variables

- **Write the following:**

```
/* A function that adds two numbers and returns the result */
function Add(n1_:Number , n2_:Number):Number
{
            return n1_ + n2_;
}

/* Creating two variables used as input for the Add function*/
var nInput1:Number = 5;
var nInput2:Number = 6;

/* Calling the Add function and displaying the result */
trace("The result is: " + Add(nInput1, nInput2));
```

Example 5: Variables *(cont'd)*

- Two ***Number*** variables nInput1 and nInput2 were declared and defined.
- The keyword ***var*** specifies that they are variables and not functions like before.

  NB: The language is case sensitive, so a variable named ***nInput1*** is totally different than another variable called ***ninput1***.
- The function Add is used by having two variables as arguments, while in the previous example the arguments were constants.
- In this case, n1_ and n2_ variables inside the Add function will have the value of nInput1 and nInput2 respectively.

Example 5: Variables *(cont'd)*

Guides to writing good code:

- Following a naming convention is really important when writing code. It makes life easier for the programmer reading the code after you.
- The important thing is to find a simple and sensible convention and to follow it religiously. It is one of the factors that will be checked for in your work.

Ex: **var nInput1:Number ;**

The name starts with a lowercase "n" since the variable is a Number variable.
Then it is easy from the name to deduce that we want to use this variable as first input.

var iNum_Points:int;

Easy to know later that it is an integer "i" and it is used to store the value of how many points we have.

A lot more on this topic will be covered in the "Types & Variables" chapter

Example 6: Arithmetic expression

- **Write the following:**

```
/* A function that computes the distance between two points */
function GetDistance(x1_:Number , y1_:Number, x2_:Number ,
                     y2_:Number):Number
{
     var nDeltaX:Number = x2_ - x1_;
     var nDeltaY:Number = y2_ - y1_;
     return Math.sqrt( nDeltaX * nDeltaX + nDeltaY * nDeltaY );
}

/* Creating four variables representing two points in a 2D space */
var nPoint1_X:Number = 2, nPoint1_Y:Number = 1, nPoint2_X:Number = 7,
nPoint2_Y:Number = 3;

/* Calling Distance function and displaying the result */
trace("The result is: " + GetDistance(nPoint1_X, nPoint1_Y, nPoint2_X,
nPoint2_Y));
```

Example 6 – Arithmetic expression *(cont'd)*

- Four variables of type Number are declared, defined and assigned a value. **var nPoint1_X:Number=2, nPoint1_Y:Number=1, nPoint2_X:Number=7, nPoint2_Y:Number=3;**
- The function GetDistance takes four Number arguments and returns a value of type Number.
- The function GetDistance requires a square root calculation which can be found under the Math Libraries **(Math.sqrt).**
- The function GetDistance declares and defines six variables: x1_,y1_,x2_,y2_,nDeltaX, and nDeltaY.
- Remember that variables declared inside a function are only available while executing the function.
- nDeltaX and nDeltaY are used to hold the difference between the first and the second point.

Example 6 – Arithmetic expression *(cont'd)*

```
var nDeltaX:Number = x2_ - x1_;
var nDeltaY:Number = y2_ - y1_;
return Math.sqrt( nDeltaX * nDeltaX + nDeltaY * nDeltaY );
```

- The statement var nDeltaX:Number=x2_-x1_; subtract x1_ from x2_ then assign the result to nDeltaX.

- The multiplication operator has a higher order of precedence than the addition operator.

- This is why nDeltaX*nDeltaX is evaluated first to 25.
- Next nDeltaY*nDeltaY is evaluated to 4.
- Then 25 and 4 are added evaluating to 29.
- Then the square root of 29 is evaluated to 5.385165 and returned by the function.

Example 6 – Arithmetic expression *(cont'd)*

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:

```
Ex:  /* Badly formatted code */
     var nBlah:Number=x1+x2*y3-y1/2;

     /* Good formatting, use spaces and parenthesis */
     var nBlah:Number = (x1 + x2) * y3 - (y1 / 2);
```

A lot more on this topic will be covered in the "Expressions, Statements and Operators" chapter

Example 7: Conditional expression

- **Write the following:**

```
/* Getting two random values between 0 and 100 */
var nNum1:Number = Math.random()*100;
var nNum2:Number = Math.random()*100;

/* Getting the sum of nNum1 and nNum2 */
var nSum:Number = nNum1 + nNum2;

/* Check if nSum's value is greater than 100 */
if(nSum > 100)
{
     trace("nNum1 + nNum2 greater than 100");
}
else
{
     trace("nNum1 + nNum2 less than 100");
}
```

Example 7: Conditional expression *(cont'd)*

- nSum>100 is the Boolean expression that evaluates to true or false,
- If the expression is true, then the statement (or the block of statements enclosed between an opening and a closing curly braces) following the condition is executed.

- If the expression is false then the statement following the condition is skipped, and the statement following the else is executed.

- The conditional statement starts with the keyword if followed by an opening parenthesis, followed by a Boolean expression, followed by a closing parenthesis followed by an instruction.

```
if(Boolean expression)
{
        instruction;
}
```

- Notice that there is no semicolon after the closing parenthesis of the Boolean expression because the conditional statement has not ended yet.

Example 8: Loops

- **Write the following:**

```
/* Declaring a variable used in the loop */
var i:Number = 0;

/* Starting a for loop from 0 to 9 and displaying the current
   value of i */
for(i = 0; i < 10; i = i + 1)
{
     trace(i);
}

/* Starting a while loop from 0 to 9 and displaying the current
   value of i */
 i=0;
 while( i < 10 )
 {
     trace(i);
     i=i+1;
 }
```

Example 8: Loops *(cont'd)*

- The for loop form is:
  for(expression1; expression2; expression3)
  {
  statement
  }

- The loop is initialized through expression1 **i=0;**

- Expression2 specifies the test made before each iteration **i<10;**

- If expression2 is true, the body of the loop will be executed followed by expression3 **i=i+1**

- The loop iterates until expression2 is false.
- If expression2 is false the for loop will exit.

- Any or all of the three for expressions may be omitted, but the semicolon must remain.

Example 8: Loops *(cont'd)*

- The while loop form is:

```
while (expression)
{
        statement
}
```

- If 'expression' **Sum<10** is true, the statement is executed until ***'expression'*** becomes false.

- In our case the statement is made from a block enclosed between curly braces:

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

- The expression is evaluated before the statement is executed,
- When the expression is false from the first time, the statement will never be executed.

Chapter 7 – 8: Conditional – Loops

Guides to writing good code:

- Always use braces and good formatting when writing a conditional expression or a loop

Ex:

Bad Code:

```
while(i<10)
{
trace(i);
i=i+1;
}
```

```
while(i<10)
trace(i);
i=i+1;
```

Good Code:

```
while( i < 10 )
{
    trace(i);
    i = i+1;
}
```

A lot more on this topic will be covered in the "Conditionals" and "Iterations" chapters

Example 9: One dimensional array

- **Write the following:**

```
var i:Number = 0;
/* Creating an array of 10 elements */
var aNumbersArray:Array = Array(10);

/* Displaying the content of the aNumbersArray array on one line */
trace(aNumbersArray);

/* Initializing the content of the array */
for(i=0; i<10; i=i+1)
{
   aNumbersArray[i] = i;
}

/* Displaying the content of the aNumbersArray array on one line */
trace(aNumbersArray);

/* Displaying the content of the aNumbersArray array each element on a line */
for(i=0; i<10; i=i+1)
{
   trace(aNumbersArray[i]);
}
```

Example 9: One dimensional array *(cont'd)*

- It is a collection of variables that are referred to by the same name.
- In our case ***var aNumbersArray:Array = Array(10);*** reserved memory for 10 variables.
- Array elements are accessed through an index, in our case the index is i.
- The array element is accessed by indexing the array name. It is done by writing the index enclosed between brackets placed after the array name. arrayName[index] **e.g: aNumbersArray[i]=i;**
- The first element is accessed by index 0 **e.g: aNumbersArray[0]**
- The highest address corresponds to the last element and it is accessed by index (total number of elements – 1), in our case it is **aNumbersArray[9]**

The End ☺