What is ActionScript ?

• **ActionScript** is an ECMAScript-based programming language used for **controlling** Macromedia **Flash movies** and applications.

• Since both **ActionScript** and **JavaScript** are based on the same **ECMAScript syntax**, fluency in one easily translates to the other.

• However, the client model is dramatically different:
  - while **JavaScript** deals with **windows, documents** and **forms**,
  - ActionScript deals with **movie-clips**, **text fields** and **sounds**.

• **Flash 7** (MX 2004) introduced **ActionScript 2.0**, which adds **strong typing** and object-oriented features such as explicit class **declarations**, **inheritance**, **interfaces**, and **encapsulation**.

• ActionScript code is frequently **written directly in the Flash** authoring environment, which offers useful reference and powerful aids for syntax checking.

• In this case, the **source code** is saved along with the rest of the movie in a **.fla file**.

• It is also common for **ActionScript code** to be imported from **external text files** via **#include statements**.

Naming

• **Naming** involves **capitalisation of code elements**.

• **Function names** and **variables** should begin with a **lower-case letter**;

• **objects** should be **capitalized**.

• The **first letter** of each **subsequent word** should also be **capitalised** in both cases.

for example:

Components or objects: *__P__roduct__I__nformation, __M__ovie__C__ontroller*

Variable or property: *user__N__ame, my__H__tml, raw__X__ml*

The Flash code editor features code completion only when variables are named according to a specific format. This involves appending the variable type to the end of the variable name.

Naming

code hints

• The Flash code editor features **code completion** only when variables are named according to a specific format. This involves appending the **variable type to the end of the variable name**.

• Supported suffixes for code completion (not completed)

Object type	Suffix string	Example
Array	_array	myArray_array
Button	_btn	myButton_btn
Color	_color	myColor_color
ContextMenu	_cm	myContextMenu_cm
ContextMenuItem	_cmi	myContextMenuItem_cmi
Date	_date	myDate_date
Error	_err	myError_err
LoadVars	_lv	myLoadVars_lv
LocalConnection	_lc	myLocalConnection_lc
MovieClip	_mc	myMovieClip_mc
MovieClipLoader	_mcl	myMovieClipLoader_mcl
SharedObject	_so	mySharedObject_so
Sound	_sound	mySound_sound
String	_str	myString_str
TextField	_txt	myTextField_txt
Video	_video	myVideo_video
XML	_xml	myXML_xml
XMLNode	_xmlnode	myXMLNode_xmlnode
XMLSocket	_xmlsocket	myXMLSocket_xmlsocket

Variables

Scoping and declaring variables

• A variable’s scope refers to the area in which the variable is known and can be referenced. There are three types of variable scopes in ActionScript:

• **Timeline variables** are available to any script on that Timeline.

• **Local variables** are available within the function body in which they are declared (delineated by curly braces).

• **Global variables** and functions are visible to every Timeline and scope in your document.

• ActionScript 2.0 classes also support **public**, **private**, and **static** variable scopes.

• Difference between the _global and _root scopes:
    • The **_root scope is unique** for each loaded SWF file.
    • Use the _global identifier to create global objects, classes, or variables.
    • The **global scope applies to all Timelines** and scopes within SWF files.

• Use **relative addressing** rather than references to _root timelines, because it **makes code reusable** and portable.

Variables

Timeline Variables

• **Timeline variables** are available to any script on that Timeline.

• To declare Timeline variables, initialize them on any frame in the Timeline.

- • Be sure to initialize the variable before trying to access it in a script. For example, if you put the code var x = 10; on Frame 20, a script attached to any frame before Frame 20 cannot access that variable.

Variables

Local Variables

• To declare local variables, use the var statement inside the body of a function.

• A local variable **is scoped to the block** and expires at the **end of the block**.
• A local variable not declared within a block expires at the **end of its script**.

• Local variables can also help **prevent name conflicts**, which can cause errors in your application.

  • For example, if you use name as a local variable, you could use it to store a user name in one context and a movie clip instance name in another; because these variables would run in separate scopes, there would be no conflict.

• It's good practice to use local variables in the body of a function so that the function can act as an **independent piece of code**. A local variable is only changeable within its own block of code. If an expression in a function uses a global variable, something outside the function can change its value, which would change the function.

For example, the variables i and j are often used as loop counters. In the following example, i is used as a local variable; it exists only inside the function makeDays():

```
function makeDays() {
  var i;
  for( i = 0; i < monthArray[month]; i++ ) {

    _root.Days.attachMovie( "DayDisplay", i, i + 2000 );

    _root.Days[i].num = i + 1;
    _root.Days[i]._x = column * _root.Days[i]._width;
    _root.Days[i]._y = row * _root.Days[i]._height;

    column = column + 1;

    if (column == 7 ) {

      column = 0;
      row = row + 1;
    }
  }
}
```

Variables

Global Variables

• Global variables and functions are **visible to every Timeline and scope** in your document.

• To create a variable with global scope, use the **_global identifier** before the variable name, and **do not** use the var = syntax.

• For example, the following code creates the global variable myName:

```
var _global.myName = “George”; // syntax error
_global.myName = “George”;
```

• However, if you initialize a local variable with the same name as a global variable, you don’t have access to the global variable while you are in the scope of the local variable:

Variables

• Variables are used **to store** a variety of different types of **information**

• A variable’s datatype relates to the information the vari-able stores and assists Flash in determining which actions are appropriate to invoke on this information.

• **Strings, Numbers** and **Booleans** are the most often used variable data types.

Variables

Numbers

• Numeric variables in Flash are variables whose values can be manipulated using mathmetical expressions like multiplication. For instance, if we wished to record the year in which something happend, we would generally do it as a number, in the following form:

```
founding_year = 2000;
current_year = 2003;

operational_for = current_year - founding_year;
```

other example:

```
my_sum = 1 + 2 + 3;
```

Variables

Strings

• Strings are **one or more character**s (letters, digits, spaces, etc.) tied together. In general this includes things like names, addresses, and other information which can't be manipulated in the same way as numbers.

• String values are signified in Flash by enclosing the text we wish to store in our variable in **double** or **single quotes**.

```
country = ‘Australia’;
country = “Australia”;

my_sum = “1 + 2 + 3”;       NO Calculation!
```

link:

Type in the box below, the code you would use to define a new variable 'name' with the value "Joe".

Test Me   Show Me

Variables

Boolean

• Boolean variables might be a foreign concept to you if you have never programmed before. Boolean variables store one of two values: **true** or **false**.

• When you get into more complicated ActionScript such as conditionals and loops you will use Boolean values often.

• They are used to store a **logical representation** of either true or false which can be used in making decisions. If you wanted to create a Boolean variable which shows that you like Flash, you would do so as follows:

```
likeFlash = true;
```

Variables

Changing Variable Types

• For instance, the following code, which creates a variable, assigns it a string value, then overwrites that value with a numeric value, is perfectly acceptable:

```
fav_color = "purple";
fav_color = 1;
```

• This flexibility is often very useful but it can also make tracking down bugs caused by assigning variables the wrong type of value very difficult. For instance, if you accidentally assign a variable a string value and then try to add it to a numeric value, Flash makes both values Strings and concatenates them together. In a small script this isn't such an issue, but in a full Rich Internet Application it can be a bug finding nightmare.

```
myText = "purple";
myNumber = 1;

myResult = myText + myNumber;
myResult = “purple1”;
```

Variables

Strict Data Typing

• ActionScript 2.0 in Flash MX (Pro.) 2004 introduced the concept of Strict Data Typing which allows developers to specify the single acceptable type for a variable when the variable is declared/defined.

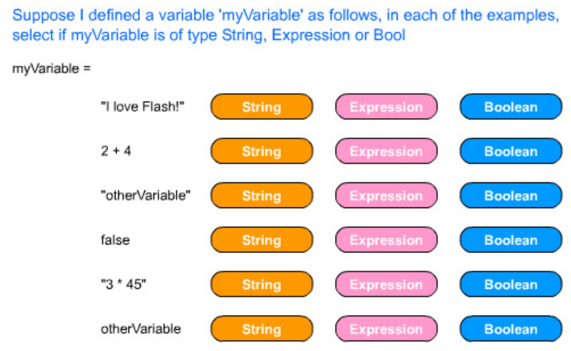
```
var fav_color:String = “purple”;
```

• Strict Typing can also be applied to function return types and arguments:

```
function doSomething(word:String):Number {
}
```

This defines a function which must take a String argument and must return a Number.

link:



Suppose I defined a variable 'myVariable' as follows, in each of the examples, select if myVariable is of type String, Expression or Bool

myVariable =

"I love Flash!"	String	Expression	Boolean
2 + 4	String	Expression	Boolean
"otherVariable"	String	Expression	Boolean
false	String	Expression	Boolean
"3 * 45"	String	Expression	Boolean
otherVariable	String	Expression	Boolean

Commenting Code

• **Commenting code** is always recommended.

• Comments should document the decisions made while building the code, **telling the story of what it attempts to do**.

• **A future developer** should be able to pickup the logic of the code with the assistance of the comments.

```
var clicks = 0;    // This is a simple comment
```

```
/*
This is a multiline comment.
.....
.....
*/
```

Commenting Code

• Some common methods for indicating important comments are:

```
// :TODO: more work to be done here
// :BUG: [bugid] this is a known issue
// :KLUDGE: this bit isn’t very elegant
// :TRICKY: lots of interactions, think twice before modifying
```

Timeline layout

• Don’t use default layer names (Layer 1, Layer 2, etc.), provide your own **intuitive labels.**

• **Groups layers together in folders**, where it makes sense.

• Place **ActionScript layers at the top of the stack**, to easily locate all the code on the timeline.

• **Lock layers currently not in use**.

Paths to Objects and Variables

• A path in Flash is analogous to a 'path' in a park; it's a way of **getting from one place to another**.

• A path in Flash directs Flash's interpreter to a specific object within the movie.

• Paths are vital to your understanding of Actionscript because ,without paths your entire SWF contents would have to reside in one place (on one level for instance), which is increasingly impossible with the fancy stuff designers and developers are doing these days.

Paths to Objects and Variables

Path to Objects

• **Common Objects** include **MovieClips**, **Buttons** and **Text-Field** but those of you who understand **Object Oriented Programming** will know that Objects in Flash are much more universal than these few examples.

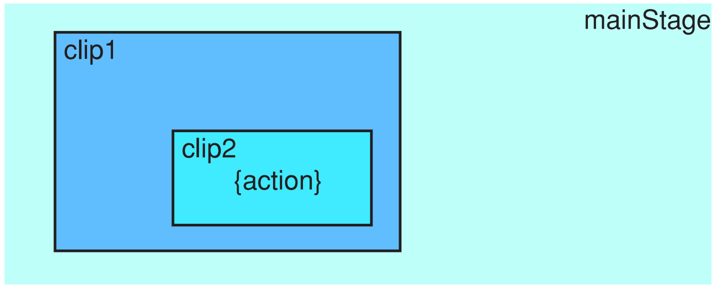
• While you're a beginner, paths are probably most applicable in terms of **Buttons** and **Movie Clips**, and using paths to reference more advanced Objects works just the same, so we will focus on these objects to begin with.

Paths to Objects and Variables

Path to Objects

Example:

1. Imagine you have a movie clip on your main stage, called 'clip1' for example. Within this clip is another clip called 'clip2'. Now imagine you have a button on the main stage also. When the button is clicked you want it to perform an action on 'clip2', (which action is not important, we'll use a visibility set in our example).



2. Selecting your button you add the actions as follows:

```
on (release) {
        setProperty (“clip2”, _visible, false);
        // OR
        clip2._visible = false;
}
```

Paths to Objects and Variables

Path to Objects

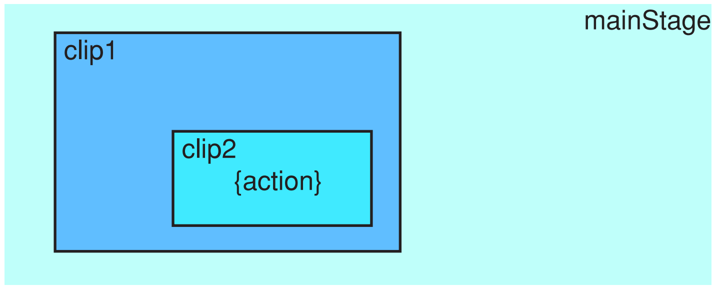
Now you test your movie, hit your button and Wham!
... Nothing happens.

Paths to Objects and Variables

Path to Objects

Why?

1. Thinking back to our example, recall that clip1 is on the main stage and it contains clip2.



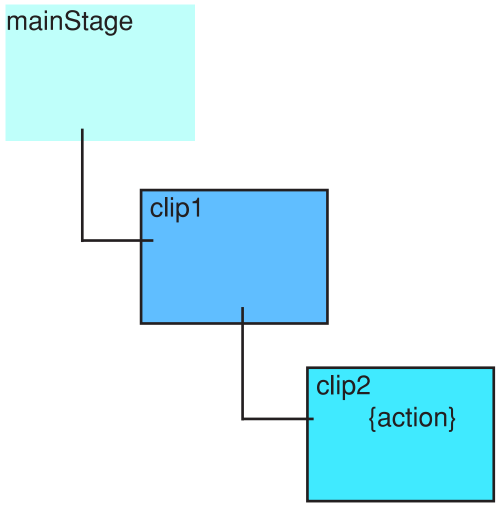
Paths to Objects and Variables

Path to Objects

2. Now, since clip1 contains clip2, clip2 does not actually reside on the **main timeline**.

It is on the timeline within the clip1 Movie Clip.

We all know this, but Flash doesn't, so we have to tell it using paths in our syntax!



Paths to Objects and Variables

Path to Objects

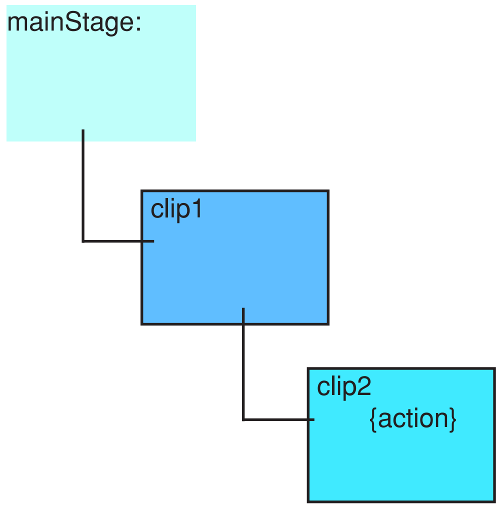
3. The table below shows the path syntax parameters and gives examples of how to use them

Parameter	**Usage**	**What it tells Flash**
_level0	setProperty ("_level0"	We're going right back to the lowest Level of the movie.
_root	setProperty ("_root"	"We're going right to the bottom of the current Level; to the Main Stage on which our base Objects are stored."
. (dot)	setProperty ("_root.clip1.clip2"	"Whatever comes after this dot further defines the path to something." In this case, clip2 is on the timeline of clip1 which is on the _root.

Paths to Objects and Variables

Path to Objects

1. ***clip1*** is on the main timeline.
2. ***clip2*** is within ***clip1***.
3. The Main Timeline can be reference using ***"_root"***.



So, the path to clip2 is:

_root.clip1.clip2

Paths to Objects and Variables

Path to Variables

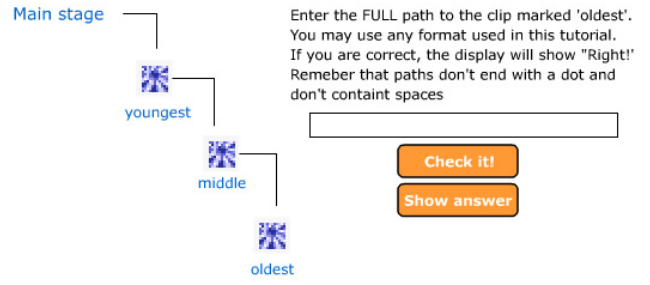
• Variables are just a **type of Object**, so paths to variables are specified in exactly the same manner as has been described before.

• For instance, to set the value of a variable called ‘foo’ within clip2 in the ongoing example:

So, the path to the Variable in clip2 is:

_root.clip1.clip2.foo = “Some Value”;

link:



Enter the FULL path to the clip marked 'oldest'.
You may use any format used in this tutorial.
If you are correct, the display will show "Right!"
Remeber that paths don't end with a dot and don't containt spaces

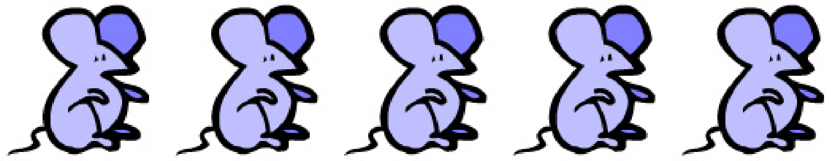
Instances

• An instance is ***a copy*** of any symbol from your Flash file's library which resides on the stage. So **any graphic**, **button** or **movie clip** you put on the stage **is an instance**.

• **All instances have Instance Names**, which by default are "instance1", "instance2", etc.

• In MX we can address buttons using their instance names and do all sorts of fancy stuff, so they've become even more important!

link:



How to learn ActionScript?

- **do** webpages ands animations with **Flash** and **ActionScript**
- **make mistakes**
- **make mistakes**
- **make mistakes**
- **make mistakes**

The Grammatik of ActionScript

Case Sensitivity

• ActionScript **is case sensitive**. This means that you have to take care about uppercase and lowercase characters.

```
APFEL != apfel != Apfel
```

The Grammatik of ActionScript

speaking Variables

- use **speaking Variables** in your Scripts
  - it makes live more easy
  - it makes your Script more **readable**
  - you and your Colleagues will understand your script faster

  - e.g. rectLength, rectMovieClipXpos

Repetition Statements

The FOR Statement

• for ... *to... do*

```
for(init; condition; next) {
        statement(s);
}
```

init: An expression to evaluate before beginning the looping sequence; usually an assignment expression. A var statement is also permitted for this parameter.

condition: An expression that evaluates to true or false. The condition is evaluated before each loop iteration; the loop exits when the condition evaluates to false.

next: An expression to evaluate after each loop iteration; usually an assignment expression using the increment (++) or decrement (--) operators.

statement(s): An instruction or instructions to execute within the body of the loop.

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

```
output: 1 2 3 4 5 6 7 8 9
```

```
for (i=10; i>1; i--){
        trace(i);
}
```

```
output: 10 9 8 7 6 5 4 3 2
```

• see also: **break;**
continue;

Repetition Statements

The DO ...WHILE Statement

• **do ...** ***while***

```
do {
        statement(s)
} while (condition)
```

condition: The condition to evaluate.

statement(s): The statement(s) to execute as long as the condition parameter evaluates to true.

```
var myVar:Number = 0;
do {
        trace(myVar);
        myVar++;
} while (myVar<5);
```

```
output: 0 1 2 3 4
```

• see also: **break;**
continue;

Repetition Statements

The WHILE Statement

• **while ...**

```
while(condition) {
        statement(s);
}
```

condition: The condition to evaluate.

statement(s): The statement(s) to execute as long as the condition parameter evaluates to true.

```
var i:Number = 0;
while (i<20) {
        trace(i);
        i += 3;
}
```

```
output: 0 3 6 9 12 15 18
```

• see also: **break;**
continue;

Repetition Statements

The FOR..IN Statement

• **for..in**

```
for(variableIterant in object) {
        statement(s);
}
```

variableIterant: The name of a variable to act as the iterant, referencing each property of an object or element in an array.

object: The name of an object to be iterated.

statement(s): The statement(s) to execute as long as the condition parameter evaluates to true.

```
var myObject:Object = {name:"Tara", age:27, city:"San Francisco"};
for (var name in myObject) {
        trace("myObject."+name+" = "+myObject[name]);
}

//output
myObject.name = Tara
myObject.age = 27
myObject.city = San Francisco
```

Conditional Statements

The IF Statement

• **if ...** ***then ..***

```
if(condition) {
            statement(s);
}
```

condition: The condition to evaluate.

statement(s): The statement(s) to execute as long as the condition parameter evaluates to true.

```
if(name == “Erica”){
            play();
}
```

Conditional Statements

The IF Statement

• **if ...** ***then ... else***

```
if (condition){
        statement(s);
} else {
        statement(s);
}
```

condition: The condition to evaluate.

statement(s): The statement(s) to execute as long as the condition parameter evaluates to true.

```
if (age>=18) {
        trace(“welcome, user”);
} else {
        trace(“sorry, junior”);
}
```

Conditional Statements

The SWITCH Statement

• **switch ()**

```
switch (expression){
        caseClause:
        [defaultClause:]
}
```

expression: Any expression.

caseClause: A case keyword followed by an expression, a colon, and a group of statements to execute if the expression matches the switch expression parameter using strict equality (===).

defaultClause: A default keyword followed by statements to execute if none of the case expressions match the switch expression parameter strict equality (===).

```
switch (String.fromCharCode(Key.getAscii())) {
case “A” :
  trace(“you pressed A”);
  break;
case “a” :
  trace(“you pressed a”);
  break;
case “E” :
case “e” :
  trace(“you pressed E or e”);
  break;
default :
  trace(“you pressed some other key”);
}
```