CGT 353: Principles of Interactive and Dynamic Media
Scripting Basics and Refresher

Introduction:

- Variables are basically treated pretty much the same in AS 2.0 and 3.0

- Four types of variables:

    - **Local** – used to track data temporarily
    - **Instance** – variable attached to a particular object within class definitions
    - **Dynamic instance**
    - **Static**

Actionscript 2.0/ 3.0

Strict or Strong Data Typing: lets you explicitly declare the object type of a variable when you create it

- Because data type mismatches trigger compiler errors, <u>strict data typing helps prevent you from assigning the wrong type of data to an existing variable.</u>

- The advantage is that if the variable type is based on a built-in class, such as Button, Array, and so on, the ActionScript editor displays code hints for the variable.

```
var sProduct = "Flash 8";
sProduct = 6;
```

```
var sProduct:String = "Flash 8";
```

After sProduct has been declared as String, if you attempt to assign a number value to it, Flash will throw an error. For example, if you attempt this:

```
sProduct = 6;
```

Then you will see the following error in the Output panel:

```
**Error** Scene=Scene 1, layer=Layer 1, frame=1:Line 2: Type mismatch in assignment statement: found Number where String is required.


sProduct = 6;

Total ActionScript Errors: 1 Reported Errors: 1
```

Be sure to only use the `var` keyword when you are first declaring the variable. If you use the var keyword subsequently, then Flash will think you are declaring a new variable. It will create a new variable with the same name, overwriting the original.

External Authoring of Scripts:

- Can author Flash scripts outside of Flash as basic text files with .as extensions.
- Can import them using "Import from File" option.
- Can also use the **#include** action to include files "on the fly."

Advantages to external scripting include:

a. Being able to use another script editor.

b. Allowing multiple programmers to work together.

c. Setting up reusable script files for specific functionalities.

Precedence:

- In object-oriented programming, if two objects exist at the same time and both are able to react to the same event, only one will.
- Which object reacts depends on a hierarchal precedence (order) of messaging.

General Precedence Rules in ActionScript:

Mouse Clicks:

- mouseDown and mouseUp movieclip events have precedence over press and release events of a button.
- All four will work, but movie clip actions will happen first.
- If a movie clip is inside a button, these won't work at all.
- Also, buttons only react when clicked on while movie clips react no matter where you click.

Button Presses:

- keyDown and keyUp are global, which means that they will react anytime a button is pressed.

- Movie clip key events supercede button key events

EnterFrame:

- Frame actions supercede movie clip actions assigned with the enterFrame event.

- However, frame actions only execute when the keyframe they are assigned is encountered.

- Movie clip actions assigned with enterFrame will repeated execute as long as the movie clip remains on the stage and the playhead is moving.

Multiple Movie Clips:

- If several movie clips share an event the order of priority is determined by the arrangement.

- On multiple layers, layer order determines precedence.

Variables:

- **Variables** are nothing more than storage names and places for data.

- **Three main concerns with Flash variables are:**

  1. the name
  2. the scope
  3. the type of data inside

Flash Variable Naming Conventions:

- All characters must be a letter, number, underscore(_), or a dollar sign ($)

- Cannot use other characters such as (*) or (/)

- Cannot be any inherent ActionScript words

- All names must be unique within their scope:

- **Local –** accessible in that function, declared with the var keyword (Ex. var members:Array = new Array();)
- **Timeline –** accessible in any function in a timeline, declared with the var keyword in a frame on timeline
- **Global –** accessible in any timeline in a movie, declared using the global identifier WITHOUT the var keyword (Ex. _global.myName = “Kellen”;), cannot specify a data type

Data Types:

- As said before, while you have to declare data types in many other languages.
- **This is not the case in Flash AS 1.0 (but is in AS 2.0 and 3.0)**
- Usually only have to worry about whether a variable is a string or a number.
- Flash uses the following data types:
  1. **Strings -** sequences of alphanumeric characters and punctuation marks identified by quotation marks
  2. **Numbers -** double precision floating point numbers on which you can perform mathematical calculations
  3. **Boolean -** binary data type being either 0 (false) or 1 (true)
  4. **Null -** used to indicate no data
  5. **Undefined -** represents a variable that has not been assigned a value
- **Objects and movie clips** are variable data types, even though they are not in the traditional sense.
- Since they contain properties that can store data, they are considered a form of data type.

When Working With Variables…

- **var** - declares a variable
- **set variable** - sets a variable equal to some value
- **delete** - deletes a specified variable to save memory

- **with()** - allows you to send a series of code statements to a single location

Operators:

- Using an English metaphor, if <u>variables are the nouns</u> of programming languages, <u>operators can be seen as verbs</u>.

- **Operators** are essentially symbols used to denote the type of operation performed on one or more operands, depending on what you are trying to do.

- **Operands** are the elements operators act upon.

- Specific operators apply to specific data types such as strings and numbers.

- **Compound operators** represent shorthand methods of performing certain tasks

Note: Always Keep Operator Precedence into Account:

- Multiplication before addition, parenthesis over multiplication., etc…

Basic Types of Operators:

1. **Number (Numeric) –** (+,*,/,%,-,++, --) 2*3, 3+4, etc
2. **String** - act on strings
3. **General**
4. **Comparison –** (<,>,<=,>=) if (score > 100)
5. **Logical -–** (&&, ||, !)
6. **Bitwise –** (&,|,<<, etc)

General Operators:

- Should already be familiar with these:

    1. **String delimiters -** sets of double quote marks used to signify string data element

    2. **Parentheses -** used to offset arguments sent to functions or methods

    3. **Dot Access Delimiter (.) -** used in referencing and targeting

    4. **Curly Brackets ({}) -** used to denote logical code groupings

    5. **Square brackets -** used to access or assign specific array values

Equality vs Assignment Operators:

myvar **=** 1	(assignment)
myvar == 1	(equality - does this equal that?)
===	(strict equality)
!=	(inequality)
!==	(strict inequality)

Logical Operators:

- Used to join (concatenate) comparison statements.
- Include:
    - **&&** (if this and that are true)
    - **||** (if this or that is true)
    - **!** (if this is not true)

Comparison Operators:

- Used to compare elements and frequently used in conditional statements
- Result of these comparisons is a binary value of either true (1) or false (0)

Special Escape Sequences:

When you are creating expressions with strings or string literals, there are special characters to use to get certain results called **escape sequences**.

\b - backspace
\f - form feed
\n - line feed
\r - carriage return
\t - tab
\" - double quote marks
\' - single quote marks

Conditionals:

- Being able to respond to certain conditions allows you to control the program flow.

- Two type of flow control statements are **conditionals and loops**

If…else/ else if

```
kellenClip.mc.onpress = function() {
        if Kellen_numb<5){
        get URL (http://www.purdue.edu","_blank");
    }  else if Jamie_numb<5{
        get URL (http://www.cgvisions.com","_blank");
    }  else {
        get URL (http://www.somebodystopme.com", "_blank");
    }
}
```

What would happen if…

```
if (4){
        trace("Hello class!");
}

or

if ("hello){
        trace("Hello class!");
}
```

- All nonnumeric strings are converted to false when used in a Boolean context.

- All nonzero numbers are converted to true.

Ternary Operator (? :)

- A shorthand way for simple conditionals is to use the ternary operator (? :)

  Basic format: condition ? do_this_if_true : do_this_if_false

Example:

```
_framesloaded <50 ? gotoAndPlay(5) : gotoAndPlay(8)
```

Switch…case/default:

- Allows you to test for any number of conditions and then respond to each.
- Useful for when you have <u>many conditions.</u>
- Break is not needed in the <u>default</u>, but needed at any other point

Example:

```
switch (number_variable) {
        case 10:
                get URL (http://www.cgvisions.com","_blank");
                break;
        case 15:
                get URL (http://www.cgvisions.com","_blank");
                break;
        case 20:
                get URL (http://www.cgvisions.com","_blank");
                break;
        default:
                get URL (http://www.cgvisions.com","_blank");
}
```

Can also use other values other than a single number:

```
switch (true) {
        case (x<10):
                get URL (http://www.cgvisions.com","_blank");
                break;
        case (x<= 10 && x <20):
                get URL (http://www.cgvisions.com","_blank");
                break;
        default:
                get URL (http://www.cgvisions.com","_blank");
}
```

Loops:

- Allow you to perform an operation for a specific set of times.
- Nearly always use a **counter**, otherwise you get stuck in an infinite loop.

Loop Terminology:

- **Initialization** - statement that defines one or more variables used in the test expression of a loop
- **Test expression** - condition that must be met in order for substatements in the loops body to be executed (conditional, test, control)
- **Iteration** - one complete pass through the loop
- **iterator or index variable** - the variable that serves as a counter in each loop

while/do…while:

- Body will be skipped entirely if loops condition not met the first time its tested

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

Example:

```
myClip_mc.onMouseDown =  function() {
        foo = 0;
        while(foo < 5) {
                duplicateMovieClip("_root.flower", "mc" + foo, foo);
                setProperty("mc" + foo, _x, random(275));
                setProperty("mc" + foo, _y, random(275));
                setProperty("mc" + foo, _alpha, random(275));
                setProperty("mc" + foo, _xscale, random(200));
                setProperty("mc" + foo, _yscale, random(200));
                foo++;
        }
}
```

for/ for….in:

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

for (i=0; i<10; i++) {
    array [i] = (i + 5)*10;
    trace(array[i]);
}
```

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

The following is an example of using for..in to iterate over the properties of an object:

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

The output of this example is as follows:

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

Can also have multiple iterators in for loops…

```
for (var i = 1, j = 10; i <= 10; i++, j--){
    trace("Going up "+ i);
    trace("Going up "+ j);
}
```

break and continue:

- The **break** command appears within a loop (for, for..in, do while or while) or within a block of statements associated with a particular case within a switch action.

- The break action instructs Flash to skip the rest of the loop body, stop the looping action, and execute the statement following the loop statement.

- When using the break action, the Flash interpreter skips the rest of the statements in that case block and jumps to the first statement following the enclosing switch action.

- Use the break action to break out of a series of nested loops.

Example:

```
i = 0;
while (true) {
        if (i >= 100) {
                break;
        }
        i++;
}
```

- **Continue** aborts a loop just like break, but unlike break, it resumes the loops execution with the next natural cycle of the loop

- **continue** appears within several types of loop statements; it behaves differently in each type of loop.

- **In a while loop**, continue causes the Flash interpreter to skip the rest of the loop body and jump to the top of the loop, where the condition is tested.

- **In a do while loop**, continue causes the Flash interpreter to skip the rest of the loop body and jump to the bottom of the loop, where the condition is tested.

- **In a for loop**, continue causes the Flash interpreter to skip the rest of the loop body and jump to the evaluation of the for loop's post-expression.

- **In a for..in loop**, continue causes the Flash interpreter to skip the rest of the loop body and jump back to the top of the loop, where the next value in the enumeration is processed.

Infinite Loops?

- Loops are limited to 15 seconds….(or more depending on publish settings)

- More than that, an alert box will pop up.

Timeline and Clip Event Loops:

- Can execute a block of code an infinite number of times without an error…
- Can execute a block of code that requires a stage update between loop iterations.
- Traditional loop statements cannot be used to perform repetitive visual or audio tasks because the task results aren't rendered before each loop iteration…

Movie Clip on Enterframe() loops

- An event handler can be used in much the same way as a timeline loop but with better results.
- The onEnterFrame() event handler causes a block of code to execute once per tick of the frame rate.
- Allows for stage updates just like a timeline loop.
- To stop an onEnterFrame loop either remove the clip that contains it or use the **delete** keyword to remove the handler.
- Be careful because timeline and onEnterFrame loops are tied to the framerate of the movie, not actual time.
- Not only do these loops make it difficult to change the frame rate later, but the end users computer will vary the frame rate playback

setInterval()

Alternative that executes a function every n milliseconds

```
moveBall  = function(){
        _root.ball_mc._x += 10;
}

ballMoverID = setInterval (moveball, 20)
```

To stop it later, write:

clearInterval(ballMover ID);