CGT 353: ActionScripting
Creating and Calling Functions

A Bit about Procedural Programming:

- Early programming consisted of one single entity: <u>the main routine.</u>

- As programs became more complex, this method became impractical.

- Besides length, the fact that the same code was often used over and over made it even more impractical.

- The answer to this problem was in the creation of **procedures.**

- Also called **subroutines** or **functions,** procedures were a way of grouping together blocks of code where execution is deferred until invoked from the main code.

- Programming that uses procedures is called **procedural programming.**

This type of programming also has its limitations, which is why object-oriented was created (more on that later…)

Advantages of Procedural over Unstructured Programming:

1. <u>More readable</u> because of less clutter and redundant coding

2. <u>More efficient</u> through the use of reusing procedures rather than retyping code

3. Procedures become a <u>centralized point for making changes</u> (think CSS…)

4. Well-written procedures <u>can be re-used</u> through many different programs

Beginning with Functions:

- Different languages use different terminology, but for ActionScript we use the term **functions.**

- **Functions** can serve as <u>subroutines</u> that simply break up the main routine and help avoid redundancy

- In these cases, no values are returned from the function.

- Can also, using the ***return*** keyword, return a value from the point it was invoked

- Functions can also accept values in the form of **parameters or arguments**

- Passing arguments allows for greater portability, which is to say that the function has a greater chance of being reused again in another program

- Functions do not always necessitate passing parameters

- Best to think of functions as "black boxes" that perform a particular operation

In addition to returning values, functions can also accept values…known as **parameters or arguments**

```
function functionName():datatype {
statements
}

function displayGreeting():Void  {
Trace(“Hello”);
}

displayGreeting();
```

Types of Functions:

1. **Functions as Subroutines:** do not return a value but rather effect something like moving a clip or invoking a trace action

2. **Functions as Data:** when functions return a value

3. **Functions within Functions:** Calling a function from within a function

4. **Recursive Functions:** when a function calls itself

Flash provides three basic types of predefined functions (or methods): **global, conversion, and mathematical**

Global Functions - designed to perform specialized tasks with data.

escape() and unescape() - used for encoding and decoding strings to URL encoded formats that escape all alphanumeric characters with various hexidecimal sequences

eval() - forces Flash to evaluate the content of a function before executing the rest of the code in which its contained thereby allowing you to dynamically generate names of things "on the fly"

Example Code: eval("card"+i) = 4

Hardwired Equivalent: Card1 = 4

getProperty() - allows you to retrieve Flash properties

Generic format: getproperty (target, property);

getTimer() - allows you to retrieve the time that has elapsed in the Flash movie, returns a numerical value

getVersion() - returns the current version of the Flash Player along with the platform the player is running on - handy for detection scripts

targetPath() - allows you to retrieve the target path to an object specified by the argument

- Used in conjunction with *trace*, you can determine the absolute path to an object

```
trace(targetpath(this));
```

Conversion Functions:

Boolean() - converts a specified value to a Boolean result, where values of either true or false are returned

```
var x=5
var myresult = Boolean(x==10)
Trace(myresult)
```

Number() and String() - allows you to convert between strings and numbers

Array() - converts data to an array

Object() - converts data to a custom object

Mathematical Functions:

IsFinite() and isNan() - evaluates data to determine if they are finite numbers or numbers at all

ParseFloat() - converts data to a floating point number

ParseInt() - converts data to an integer

Defining Custom Functions:

- As we have seen, there are a number of <u>built-in functions</u> within ActionScript.
- But these only allow you to do certain things…
- To be able to be truly fluent at any programming language, you must master the <u>creation of custom functions.</u>
- When you do this keep these things in mind:
    - Function names follow the same rules as variables.
    - All functions must be declared using the *function* keyword
    - All function definitions must include a pair of parenthesis immediately before the function body.
    - The body must always be defined by an opening and closing curly brace ({})
    - Functions can return a single value that is done by the use of the **return** keyword
- There are two ways of defining a function, the first of which creates a **named function.**
- A named function means it can be referred to by name within ActionScript:

```
function functionName (parameter1:datatype,
parameter2:datatype):dataType{
        FunctionBody
}

function circleArea(radius:Number):Number {
        return MATH.PI*(radius*radius);
}
```

```
function move(x:Number,y:Number,myMC:Moveiclip): Number {
        myMC.x = x;
        myMC.y = y;
}
```

- The second way of creating a function is similar to the first.

- Creates what is called an **anonymous function**, which cannot refer to itself by that name.

```
var functionName: Function = function( param1:datatype, param2datatype):Number{
    functionBody
}

var circleArea:Function = function (radius:Number):Number {
    return MATH.PI*(radius*radius);
}

var move:Function = function(x:Number,y:Number,myMC:Moveiclip):Number {
        myMCx = x;
    myMCy = y;
}
```

- There are many reasons for using one or the other, which we will discuss later

- One reason immediately worth noting is that named functions are available from anywhere within their scope, no matter if they are defined before or after they are invoked

Calling Functions:

- Unless a function is invoked (called), nothing will happen

```
function testFunction():Void {
        trace("this is a test class");
}
```

What will this write to the output window?

- The most common way to invoke a function is by simply calling it by name within your program, much like an action.

```
testFunction();
```

- The function name must always be followed by the parentheses, which together are called the **function call operator**

Passing Parameters:

- Some functions do not need any information passed to them, but some do

```
circleArea = function(radius):Number {
        return MATH.PI*(radius*radius);
}
```

- In the function above, a single parameter is passed to the function…

- To pass a value for that parameter, you would write:

```
area = circleArea(5);
```

To pass multiple parameters, you separate them with commas.

```
function formatMessage(to, from, message){
        return "this is a message to "  +  to +  ", from "  +  from  + ": " + message;
}
```

And you could call the function like so:

```
theMessage = formatMessage("me","you","hi :)");
```

Calling a Named Function:

```
writeMsg("before");

function writeMsg(message){
        trace(message);
}

writeMsg ("after");
```

WHAT WOULD THIS WRITE TO THE OUTPUT WINDOW?

```
before
after
```

Calling an Anonymous Function:

```
writeMsg("before");

writeMsg = function(message){
        trace(message);
}
writeMsg("after");
```

WHAT WOULD THIS WRITE TO THE OUTPUT WINDOW?

After

What is an Array?

- Is a composite data structure that can encompass multiple individual data values.
- Can include more than one data value, and should be viewed as a general purpose container.

Components of an Array:

- Each item stored in an array is an **element** of that array.
- Each element has a unique numeric position called an **index.**
- Like a variable each array element can store information just like a variable.
- So, an array is simply a collection of sequentially named variables.
- To manipulate values in a array, we ask for each element by number.
- Index values start at 0, not 1.
- Can have gaps in the indexing. For example, you could have an array at 0 and 5, but without 1,2,3, and 4

Creating Arrays:

- Can either create arrays with a data literal or with the array constructor function, Array()

- Usually easier to use an array literal

[expression1, expression2, expression3]

["Kellen", "Amy", "Mary", "Jane"]

- With the array constructor, you would write:

```
var KellensList = new Array ("Kellen", "Amy", "Mary", "Jane");

or

var KellensList = new Array (4);
```

Types of Arrays:

- Single dimension
- Parallel
- Associative

- **Single dimension arrays are what we have been discussing.**

- **A single dimension array simply refers to single columns of indexed data:**

```
oneArray = ["a","b","c"];
twoArray = new Array();
threeArray = new Array("a","b","c");
```

- **Parallel arrays** come from having two groups of data that are connected.

- Much easier than writing out two completely sets of one dimension arrays.

```
employees = newArray();
employees[0] = "Ty:January 10";
employees[1] = "Kellen:June 13";
employees[2] = "Kara: April 5";
```

- Then you would have to split them with the String object methods..

- Much easier to write:

```
employees = newArray("Ty", "Kellen", "Kara");
birthdays = newArray("10", "13", "5");
```

Then retrieve them by:

```
trace(employees[1] + "'s birthday is " + birthdays [0]);
```

Associative Arrays:

```
title= new Array("professor","student","staff");
person = new Array("Kellen Maicher","Joe Blow","John Doe");
```

This would be better written like this:

```
person = new Array(3);
person ["professor"] = "Kellen Maicher";
person ["student"] = "Joe Blow";
person ["staff"] = "John Doe";
```

For loops with associative arrays:

```
For (particular title){
        Trace(The " + title + " is " + person[title]);
```

Which to use: Associative or Parallel?

1. The indexes (also called keys) to an associative array must be unique.
2. Associative arrays will maintain relationships where parallel may not

Arrays as Objects:

Because arrays are objects, you can access their elements as properties of the object using the dot operator

So this…

```
myArray = new Array();
myArray["a"] = 1;
myArray["b"] = 2;
myArray["c"] = 3;
```

…..could be written like this.

```
myArray = new Array();
myArray.a = 1;
myArray.b = 2;
myArray.c = 3;
```

Multidimensional Arrays:

- **To create truly complex arrays that index values of many different data types, you create multidimensional arrays.**

```
// create the constructor for the employee objects

function Employee(nm,bday, pstn)[
    this.name = nm
        this.birthday = bday
        this.position = pstn


//create the array

employees = new Array();


//populate the array

employees[0] = new Employee ("Kellen","June 13","Professor");
employees[1] = new Employee ("Kara","June 19”,"Staff");
employees[2] = new Employee ("Don","June 22”,"Student");
```

To display the information:

```
For (i =0;  i <employees.length; i++){
        report += employees[i].name = " " + employees[i].birthday + " " +
        employees[i].position;
}
```

Array Object Methods

- **join() =** returns a string value of the elements of an array

- Used most commonly to send data from Flash to other applications

```
animals = new Array("dog","cat","bird);
strAnimals1 = animals.join();          // returns "dog,cat,bird"
strAnimals2 = animals.join(" : ");     // returns "dog : cat : bird"
```

- **concat() -** creates a new array and adds those elements to an existing array

- **slice() -** returns a new array that consists of a slice of the original array

- **push() -** adds elements to the end of an array

- **unshift() -** puts new elements to the beginning of the array and shifts the others over right

- **pop() -** allows you to remove the last element from an array and return its value

- **shift() -** removes the first element from the array, returns the value, and shifts the remaining values back one

- **splice() -** modifies the existing array by removing the number of elements from a particular element and inserting the new elements

- **sort() -** sorts elements of the array

- **sorton() -** used in parallel and multidimensional arrays to sort by a particular index

- **reverse() -** reorders the orginal array by placing the last element first and so on….

Array Summary:

- Arrays are indexed data structures in which each piece of data (**elements**) has a unique index by which it can be referenced.

- You can use the **array access operator []** to read and write from arrays.

- Different types of arrays include **basic, parallel, associative, and multidimensional.**

- The many array **methods** allow you to manipulate arrays in any number of ways.