

# CS 116 – Action Script Functions

Elie Abi Chahine

# Functions

Functions are blocks of code that carry out specific tasks and can be reused in your program.

Functions have always been extremely important in ActionScript and any other language.

# Functions

General form of a function:

```
function  FunctionName(Parameters):ReturnType
{
    statements (instructions)
}
```

- FunctionName: follows the same naming convention as a variable
- Parameters: specifies what the function expects from the user.  
Any type can be a parameter. (eg: iName:int )

**NB: You can send more than one parameter (use the “,” operator to separate them. They don’t have to have the same type.  
Leaving it empty means the function doesn’t require any parameters.**

- ReturnType: specifies if the function return anything after it executes. It can be empty, which means that the function doesn’t return anything.

# Functions

Eg:

```
function SayHello() /* doesn't take parameters. Doesn't return anything */  
{  
    trace("Say Hello");  
}
```

```
function Add(iNum1_:int , iNum2_:int):int /* takes 2 int as parameters */  
{ /* returns an int */  
    return iNum1_ + iNum2_;  
}
```

```
function ValueOfPI():Number /* doesn't require parameters */  
{ /* returns a Number */  
    return 3.1415926;  
}
```

# Calling Functions

- You call a function by using its identifier followed by the parentheses operator ( )
- You use the parentheses operator to enclose any function parameters you want to send to the function.
- For example, the **trace()** function, which is a top-level function in the Flash Player API, is used throughout this book:

***trace("Use trace to help debug your script");***

# Calling Functions

- If you are calling a function with no parameters, you must use an empty pair of parentheses.
- For example, you can use the **Math.random()** method, which takes no parameters, to generate a random number:

```
var nRandNum:Number = Math.random();
```

**NB:** if you want to know what the function does, do the following:

- Select the function
- Right Click, View Help

# Calling Functions

The function's body is only executed when the function is called and not when it is created.

**Ex:**

```
function Say_Hello_10_times()  /* the code is not executed yet */
{
    for(var i:int = 0; i<10; ++i)
    {
        trace ("Hello");
    }
}
```

```
Say_Hello_10_times();  /* now the code inside the function is executed */
```

# Parameters & Return Types

*If you declare (create) a function that takes a certain parameter type or returns a certain type, you are obliged to send or return the exact number of parameters and type*

Eg: `function Add(iNum1_:int , iNum2_:int):int`  
`{`  
`return "Hello"; /* Wrong since it expects an int to be returned */`  
`return iNum1_ + iNum2_; /* Correct since the value returned is an int */`  
`}`

**`/* Calling the Add function */`**  
**`Add ("5" , 12); /*Wrong because "5" is not an integer */`**  
**`Add (3); /* Wrong it expects 2 parameters of type int */`**  
**`Add(5,7,2); /* Wrong it expects only 2 parameters and not 3 */`**  
**`Add(5,7); /* Correct */`**



# Return

- To return a value from your function, use the return statement followed by the expression or literal value that you want to return.
- For example, the following code returns an expression representing the parameter:

```
function DoubleNum(iBaseNum_:int):int  
{  
    return (iBaseNum_ * 2);  
}
```

# Return

- Notice that the return statement terminates the function, so that any statements below a return statement will not be executed, as follows:

```
function doubleNum(iBaseNum_:int):int  
{  
    return (iBaseNum_ * 2);  
    trace("after return"); // This trace statement will not be executed.  
}
```

- Again you **must** return a value of the appropriate type if you choose to specify a return type.
- For example, the following code generates an error, because it does not return a valid value:

```
function doubleNum(iBaseNum_:int):int  
{  
    trace("after return");  
}
```

# Parameters

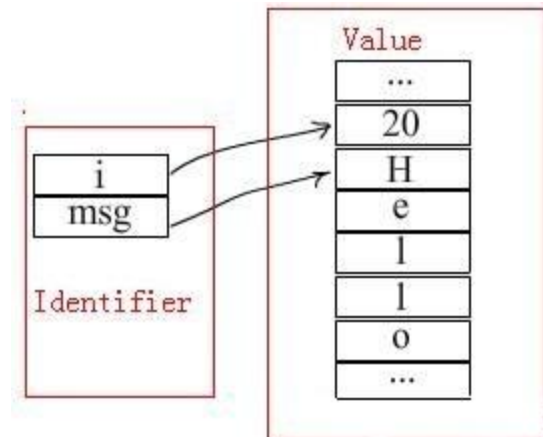
- As I already mentioned, a function can take parameters.
- Now the question is, what will happen to the variables that we send as parameters? What will happen if we change the values inside the function? Did the original variable change? ect...
- Now is a good time to start talking about the concept of passing arguments by value or by reference.
- In many programming languages, it's important to understand the distinction between them because it will affect the way you design your code (functions, classes, ect....).

# Primitive vs Complex Types

- Before I start talking about sending by reference or by value, I need to talk about the difference (in ActionScript 3.0) between “Primitive” and “Complex” types and how they affect variables.
- Primitive types are: ***Boolean, Number, int, uint, and String***
- Complex types are: **All the rest ☺ ( Array, Object, XML, MovieClip ... )**
- When we create variables (declare and assign values to them), the following is happening:

```
var i:int = 20;
```

```
var msg:String = “Hello”;
```



**Note:** This memory layout is just a simplified model for the actual memory layout in ActionScript 3.0. The actual memory layout is much more complicated than this.

# Primitive vs Complex Types

**What is the difference between primitive values and complex values?**

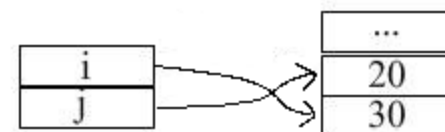
- You might say that the primitive values seem to be simple, and the complex values seem to be complicated.... True...
- You might also say they have complicated names, like XML, Date, etc... Also, true...
- The key thing is that, when you declare a primitive type variable, the reference will point to an immutable (unchangeable) object. But, if you declare a complex type variable, the reference just points to that kind of object, not immutable object.
- I know!! I know!!!! It is hard to understand without an example.... Let's do an example!!!!

# Primitive vs Complex Types

- Using primitive type variables:

```
COMPILER ERRORS ACTIONS - FRAME
1  /* declare int type variable i, value 20 */
2  var i:int = 20;
3  /* assign i to j */
4  var j:int = i;
5
6  /* Change the value of i */
7  i += 10;
8
9  trace("i = " + i); //output i
10 trace("j = " + j); //output j
11
```

```
OUTPUT ACTIONS - FRAME
i = 30
j = 20
```

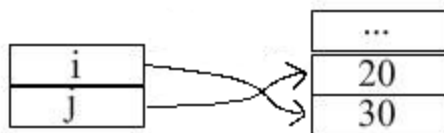


# Primitive vs Complex Types

Using primitive type variables:




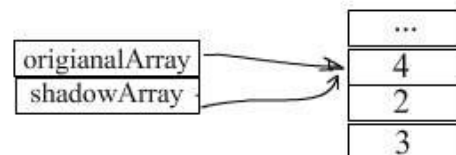
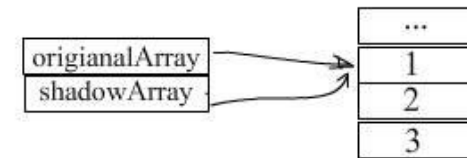
- When we changed the value of “i”, since value 20 is an immutable object, a new immutable object got generated with value 30 and let “i” point to it.



# Primitive vs Complex Types

## Using complex type variables:

```
EDITOR  TIMELINE  ACTIONS - FRAME
+ [A] [⊕] [✓] [≡] [💬] [🔗] [🔗] [🔗] [🔗] [🔗] [🔗] [🔗]
1  /* declare an array contains three item 1,2,3 */
2  var aOriginalArray:Array = new Array(1,2,3);
3  /* assign array to the aShadowArray */
4  var aShadowArray:Array = aOriginalArray;
5
6  /* change the first element of aShadowArray */
7  aShadowArray[0] = 4;
8  /* output the array */
9  trace("Values inside originalArray: " + aOriginalArray);
10 /* output the aShadowArray */
11 trace("Values inside aShadowArray: " + aShadowArray);
```



OUTPUT COMPILER ERRORS MOTION EDITOR TIMELINE

Values inside originalArray: 4,2,3

Values inside aShadowArray: 4,2,3



# Parameters

- Back to parameters and the difference with sending by reference or by value.
- Let's start with an example:

```
function passPrimitives(iXValue_:int, iYValue_:int):void  
{  
    iXValue_++;  
    iYValue_++;  
    trace(iXValue_, iYValue_);  
}
```

```
var iNum1:int = 10;  
var iNum2:int = 15;  
trace(iNum1, iNum2);           /* output: 10 15*/  
passPrimitives(iNum1, iNum2); /* output 11 16*/  
trace(iNum1, iNum2);         /* output  ????? */
```

# Parameters

- If those variables are sent by reference, then the variables sent as parameters will be changed and the output will be “11 16” .
- If they are sent by value, that means the function will create two separate variables to use inside the function, and will not touch the values of the variables that are outside. So the output will be “10 15”
- The answer is: **“10 15”**
- So in our case, the int variables are sent by value so whatever you do inside the function the original variables outside will not be touched.

# Parameters

- Another example:

```
function passPrimitives(aParamValues_:Array):void  
{  
    aParamValues_ [1] = 17;  
    trace(aParamValues_);  
}
```

```
var aValues:Array = [1,2,3,4];  
trace(aValues);                /* output: 1,2,3,4 */  
passPrimitives(aValues);      /* output: 1,17,3,4 */  
trace(aValues);                /* output: ??????? */
```

The output is going to be: “1 , 17 , 3 , 4”. Which means that the array variable was sent by reference which made the variable inside the function and the one sent as parameter both change. In other words, the function did not create it's own local variable, it used the same variable memory that was sent from outside.

# Parameters

- So now it is up to you guys, to try every type in ActionScript and tell me if it is sent by reference or by value.

**Come on, you have 5 minutes to do so!!!**

# Parameters

## Or NOT!!!

- So the answer is really simple. All primitive types are sent by value, the rest (which are the complex types) are sent by reference.
- Primitive Types: **Boolean, int, uint, Number, String, Null, and void.**
- Some Complex Types: **Object, Array, Date, XML, XMLList . . .**
- So be careful when working with functions cause you might be changing more than one variable at a time when you send it as parameter and work with it inside a function.

# Scopes

- The last thing we want to cover in this chapter is variable scope.
- The **scope** of a variable is the area of your code where the variable can be accessed.
- A **global** variable is one that is defined in all areas of your code, whereas a **local** variable is one that is defined in only one part of your code.

# Scopes

- A **global** variable is a variable that you define outside of any function, which allows everyone to access it.

Example:

```
var sGlobal:String = "I am a global variable";  
function scopeTest()  
{  
    trace(sGlobal); /* I am a global variable */  
}  
scopeTest();  
trace(sGlobal); /* I am a global variable */
```

# Scopes

- You declare a ***local*** variable by declaring the variable inside a function definition. The smallest area of code for which you can define a local variable is a function definition.

Example:

```
function localScope()  
{  
    var sLocal:String = "I'm a local variable";  
}
```

```
localScope();  
trace(sLocal); /* error because sLocal is not defined globally */
```



# Scopes

- ActionScript variables, unlike variables in C++ and Java, do not have block-level scope.
- A block of code is any group of statements between an opening curly brace ( { ) and a closing curly brace ( } ).
- In some programming languages, such as C++ and Java, variables declared inside a block of code are not available outside that block of code. This restriction of scope is called block-level scope, and does not exist in ActionScript.
- If you declare a variable inside a block of code, that variable will be available not only in that block of code, but also in any other parts of the function to which the code block belongs.

# Scopes

- For example, the following function contains variables that are defined in various block scopes. All the variables are available throughout the function.

```
function blockTest(iValue_:int)  
{  
    var iLocalValue:int = iValue_;  
    if (iLocalValue > 0)  
    {  
        var sText:String = "The Number is " + iLocalValue;  
    }  
  
    /* Even though sText was defined in the if condition, we are able to access it */  
    trace(sText);  
}  
  
blockTest(10); /* The number is 10 */
```

# Scopes

- An interesting implication of the lack of block-level scope is that you can read or write to a variable before it is declared, as long as it is declared before the function ends.
- This is because of a technique called hoisting, which means that the compiler moves all variable declarations to the top of the function.
- For example, the following code compiles even though the `initialtrace()` function for the `num` variable happens before the `num` variable is declared:

```
trace(iNum);                /* NaN */  
var iNum:Number = 10;  
trace(iNum);                /* 10 */
```

**Note:** The compiler will first take it as its default value, that is why with the `Number` type we got “NaN”.

```
iNum = 5;  
trace(iNum);                /* 5 */  
var iNum:Number = 10;  
trace(iNum);                /* 10 */
```

# Scopes

- Now, I'm going to give you multiple slides to try out and learn more about scoping.
- You need to try them out.

```
var iNum:int = 10;
```

```
function GlobalLocalTest()  
{  
    var iNum:int = 0;  
    trace(iNum);  
}
```

```
GlobalLocalTest();  
trace(iNum);
```

# Scopes

Another example to test:

```
var iNum:int = 10;

function GlobalLocalTest()
{
    trace(iNum);
    if(iNum > 0 )
    {
        var iNum:int = 0;
        trace(iNum);
    }
}

GlobalLocalTest();
trace(iNum);
```

# Scopes

Find more tests on your own and ask me questions in the  
LAB!!!

# The End 😊