Chapter 8: Functions and Scope

We have been using functions since the beginning of our adventure in ActionScript game programming and have not defined them. Now we finally give more details on functions. As we have seen, functions can be used to group together a bunch of statements into a single logical entity. Then are used for class methods. They also can be used to return values, and they definitely affect program execution flow. If used incorrectly, they can also have very hard-to-find side effects if you abuse “scope”, i.e. where variables can be seen and accessed.

8.1: Functions

Consider our first example:

```
www.cs.du.edu/~leut/1671/flashFiles/c8_function1.fla

f1 = function() {
    trace("inside f1") ;
}

f2 = function() {
    trace("inside f2") ;
}

f3 = function() {
    trace("inside f3") ;
    trace("pigs are cool!") ;
}

trace("Before calling any function") ;
f1() ;
f3() ;
f2() ;
trace("After calling the functions") ;
```

In this example we define three functions: f1( ), f2( ), and f3( ). Functions are not executed until they are called. Thus, when flash executes the first frame, the first 9 lines do nothing other than to define the functions. Nothing is printed out. The last 5 lines are then executed, and here the three functions are called. The order in which the functions are called is f1, f3, and then f2, thus, the code is executed in that order also. Thus, the output:

```
Before calling and function
inside f1
inside f3
pigs are cool!
```
Program execution simply goes from the main code in frame 1 into the code belonging to each function as it is called. If one calls the same function 3 times in a row the code in that function would be executed 3 times in a row, once for each function call.

The preceding example shows how function calls affect and resultant program execution, but they are rather silly functions. Functions are usually used to accomplish a task or calculate and return a value. An example of the former would be the updateXY( ) method in the Moving MC class. Let's consider some examples of functions that return values:

www.cs.du.edu/~leut/1671/flashFiles/c8_function2.fla

```
// This function has one input argument and returns the number * 2
DoubleTheNum = function(inputNum:Number) : Number {
    var doubledNum;
    doubledNum = inputNum * 2;
    return(doubledNum);
}

// This function returns the input argument value squared
SquareTheNum = function(inputNum:Number) : Number {
    return(inputNum * inputNum);
}

// This function returns the input argument cubed
CubeTheNum = function(n1:Number) : Number {
    return(n1 * n1 * n1);
}

// This example has two arguments. It returns the first argument raised to the second
Power = function(num:Number, power:Number) : Number {
    var returnValue:Number = 1;
    for (var i:Number = 0; i < power; i++)
        returnValue *= num;
    return(returnValue);
}

var n1:Number = 5;
var n2:Number;
n2 = DoubleTheNum(n1);
trace(n2);
n2 = SquareTheNum(n1);
trace(n2);
n2 = CubeTheNum(n1);
trace(n2);
n2 = Power(5,3);
trace(n2);
n2 = Power(5,4);
trace(n2);
n2 = Math.pow(5,4);
trace(n2);
```
Here we define four functions: DoubleTheNum(), SquareTheNum(), CubeTheNum(), and Power(). The first three functions have one input and return one value. The fourth function has two inputs and returns one value. For some reason that makes little sense computer scientists call function inputs arguments. Our functions are well named, the first returns a number equal to double the input argument. The second returns a number equal to the input argument squared, and the third function returns a number equal to the input argument cubed. The fourth function returns a number equal to the first argument raised to the power of the second argument. Let's look at the syntax of the first function:

```
DoubleTheNum = function(inputNum:Number) : Number {
    var doubledNum;
    doubledNum = inputNum * 2 ;
    return(doubledNum) ;
}
```

First we have the function name followed by the = sign, thus you are “assigning” this function to the name “DoubleTheNum”. Next is the word function followed by a left parenthesis and the argument declaration. Here one argument is declared, inputNum. We specify that this argument is type Number. We do not use the word “var” here, because we are not creating a variable, rather, the parameter that is passed in when the function is called already exists. If we have multiple arguments they are separated by commas as in the Power function above. After the closing parenthesis we specify the type of data that the function returns. In this case we see that the function returns a Number. Then we have our usual curly braces to specify the extent of the statement block. The final thing to note here is the last line of the function. This line specifies that the function should return the value that is currently in variable `doubledNum`.

Look at the code following the functions, i.e. the code that calls the functions.

```
var n1:Number = 5 ;
var n2:Number ;
n2 = DoubleTheNum(n1) ;
trace(n2) ;
n2 = SquareTheNum(n1) ;
trace(n2) ;
n2 = CubeTheNum(n1) ;
trace(n2) ;
n2 = Power(5,3) ;
trace(n2) ;
n2 = Power(5,4) ;
trace(n2) ;
n2 = Math.pow(5,4) ;
trace(n2) ;
```

The first two lines declare variables and initialize the contents of n1 to 5. The third line calls the function DoubleTheNum( ). Notice that inside the parenthesis we have variable n1. This means that the current contents of variable n1 are passed into the function. The argument `inputNum` is initialized with the value currently in variable n1. Thus, before the first line of code in function DoubleTheNum( ) is executed, the argument `inputNum` has its contents set to this value. Inside the function `inputName` can be used like any other variable. Notice also that the output of DoubleTheNum( ) is assigned to the variable n2.
The rest of the function calls work in the same way. The call to Power( ) is similar but since it has two arguments we must supply two values (or variables) inside the parenthesis when we call the function. The order of the values is important as the first is assigned to the first argument name, num in this case, and the second is assigned to the second argument name, power.

The previous example had functions that take as input Numbers and return as output Numbers, but in general functions can take in and return any type of arguments (strings, numbers, dates, arrays, or even objects belonging to user defined classes). We show examples of other types with Strings:

www.cs.du.edu/~leut/1671/flashFiles/c8_functions3.fla

DuplicateString = function (inputString:String) : String {
    var returnVal:String = inputString + " " + inputString;
    return(returnVal);
}

AppendInReverseThreeStrings = function (in1:String, in2:String, in3:String) : String {
    var returnVal = in3 + " " + in2 + " " + in1;
    return(returnVal);
}

PrintArrayOfStrings = function (bunchOfStrings:Array) : Void {
    for (var i:Number = 0 ; i < bunchOfStrings.length ; i++)
        trace(bunchOfStrings[i] + " ");
}

var s1:String;
s1 = DuplicateString("pig");
trace(s1);
var s2:String = "I";
var s3:String = "like";
var s4:String = "pigs";
s1 = AppendInReverseThreeStrings(s2,s3,s4);
trace(s1);

var manyStrings:Array = ["dog","cat","pig","mouse","cow","sheep","goat","aardvark"];
PrintArrayOfStrings(manyStrings);

manyStrings.sort();
trace("\nAfter sorting:");
PrintArrayOfStrings(manyStrings);

Here three functions are defined: DuplicateString( ), AppendInReverseThreeStrings( ), and PrintArrayOfStrings( ). The first function takes in a String argument and returns a String. The second takes input of three strings and returns a String. The third takes in an Array and does not return a value. When there is no return value, as in the third case, we use the word void to specify that the return is ignored. Look at the code that calls the function PrintArrayOfStrings( ). We can see there is no use of function output as there is in the other examples.
8.2: Scope

Now that we see how to define and call functions, the next question is how do variables inside and outside functions differ. The simple answer is variables defined inside of a function cannot be accessed outside of that function, but variables accessed in the main code can be accessed from within a function. Furthermore, it is possible to use the same variable name both inside a function and in the main code, in which case when execution is inside the function the local variable is used but otherwise the variable in the main code is used. The idea is to make functions self-contained, i.e. to keep all the code belonging to a function contained inside that function. There is no reason why code outside of a function should be accessing variable contents inside that function. Got that? Okay, maybe not, so let's consider some examples.

```
www.cs.du.edu/~leut/1671/flashFiles/c8_scope1.fla
```

```javascript
f1 = function() {
    var i:Number = 5 ; // this declares variable i, but i's scope is limited to this function
    trace("Inside f1  i = " + i) ;
    trace("Inside f1  j = " + j) ;
}

var j:Number = 8 ; // this declares variable j, and it's scope is anyplace in the file
trace("In the main code, var i = " + i) ;
trace("In the main code, var j = " + j) ;

f1() ;
trace("In the main code after calling f1(), var i = " + i) ;
trace("In the main code after calling f1(), var j = " + j) ;
```

Let's start with the code in the main part first, i.e. outside of the function. Here we declare a variable, j, and initialized it with a value of 8. Then we print out the contents of variables i and j. For variable j we get 8 as expected. Because we have not declared and initialized variable i we get “undefined” as we would expect.

We then call function f1(). Inside function f1() we declare and initialize variable i with a value of 5. We call variable i a local variable because it is defined locally to the function. We do not declare or initialize variable j. We then print out the contents of variables i and j. Variable i contains 5 as we would expect, but variable j is NOT undefined, rather, it has 8 in it. This is because if we access a variable not defined within a function the code looks for the variable in the main code. If the variable is found in the main code it is use. If not found, then it is undefined as normal.
After the program completes the execution of function f1() execution returns to the main code after the call. Here we print out the contents of variables i and j again. Again variable i is undefined. This is because the variable i is unknown outside of the function. When a variable is declared inside of a function it is only known within that function. We say the variables scope is only within the function. We can represent this pictorially as follows:

![Diagram of function f1 and main code with variable scopes]

The inner box represents the view of function f1(), and the outbox represents the view of the main code. When the execution of a program is inside a function, the variables it can access are those that are local and then any in the main code that are not also declared locally. When we are inside a box, we can see the inside or our box plus anything in the box that encloses us. BUT, the box is one-way, i.e. when we are in the outer box we can not see inside the inner box. So, when we are inside function f1(), we can access both variables i and j, but when in the main code we can only access variable j.

So what happens if a variable is declared in BOTH a function and the main code as shown in the following picture:

![Diagram of function f1 and main code with overlapping variable scopes]
In this case when we are inside f1() we access the local variable i, not the variable i in the main. Consider the following code example:

www.cs.du.edu/~leut/1671/flashFiles/c8_scope2.fla

```plaintext
f1 = function() {
    // variable i is a LOCAL variable, any changes made to variable i
    // only are visible while inside f1, this variable i no longer exists outside of the function
    var i:Number;
    i = 88;
    trace("Inside f1 variable i = " + i);
}
f2 = function() {
    // Here variable i has not been declared in the function, thus, the
    // variable being modified is the one OUTSIDE of the function
    i = 22;
    trace("Inside f2 variable i = " + i);
}
var i:Number = 5;
trace("Before call to function f1() variable i = " + i);
f1();
trace("After call to function f1() variable i = " + i);
f2();
trace("After call to function f2() variable i = " + i);
```

The first trace statement in the main code prints out value 5. Then the code enters function f1(), declares a locate variable i, assigns 88 to it, and then prints out 88. Now f1() is exited and flow moves back the main code which prints out the contents of variable i. Because the main code knows nothing about variables inside of functions it prints out the value of 5. Now function f2() is entered. Here 22 is assigned to variable i and printed out. Because we did not define variable i inside this function the code “looks outside the box” and uses the variable declared in the main code. We then exit function f2() and go back to the main and print out the contents of variable i. Here the value of 22 is printed out because the change made in function f2() was applied to the main variable i, not a local variable.

The type of the variable does not make a difference, the same scope rules apply regardless of type. Consider the following example:

www.cs.du.edu/~leut/1671/flashFiles/c8_scope3.fla

```plaintext
f1 = function() {
    var j:Number = 5;
    trace("Inside f1() var j = " + j);
}
f2 = function() {
```

```plaintext
```
var animal:String = "pig";
trace("Inside f2() var animal = " + animal);
}
f1();
trace("In _root, var j = " + j);
f2();
trace("In _root var animal = " + animal);
```

Same deal here, in the main neither variable “j” nor “animal” have been defined, hence they show up as undefined in the main code.

Finally, what happens if I declare a variable that already exists? Consider the following code:

```
www.cs.du.edu/~leut/1671/flashFiles/c8_scope4.fla

// scope only applies to functions, NOT for loops
for (var i:Number = 0 ; i < 10 ; i++) {
    trace("in loop, i = " + i);
}

// The above created variable i in _root, and it is still accessible
trace("outside of loop, i = " + i);

// but you can re-define variables, the old variable is replaced with
// the new one of the same name:
for (var i:Number = 0 ; i < 20 ; i+=5) {
    trace("hi");
}
trace("outside second loop i = " + i);
```

As you can see, outside of the for-loop the variable i is known. In the second for-loop I declare variable i a second time. What happens is the old variable is replaced by the new one.

To summarize, we have seen is that variables have a scope, that is an environment in which they can be seen. If one defines a variable inside of a function that variable can only be seen inside of that function. If a variable is defined in _root AND the same variable name is used inside of a function without declaring that variable name inside the function, then the variable in _root is used and hence changes inside the function can affect the variable outside the function. **This is bad programming practice.** It can create unintended side effects. Always make sure that you define the variables you use INSIDE the function. A function should well defined inputs, an output, and everything else should be invisible to the function user.

What about variables inside of class definitions? They are hard to get messed up, even if they are public. The reason is that you need to use the “dot” notation to access them, hence
it is very explicit what you are doing. For example, assume I have a Person class as in Chapter 7 and then have the following code:

```javascript
var name:String = "Bob";
var p1:Person = new Person(“Susan”,19);
trace(name);
trace(p1.name);
p1.name = name;
```

In this example there are two person variables, the one I defined and the one inside the Person class. Because of the “dot” notation, it is crystal clear which name variable I am accessing above.