

# ActionScript 3.0 for Designers - Lesson 1

Instructor: Frank Stepanski

## Overview

Currently ActionScript 3.0 is the internal programming language of Flash, Flex and AIR (the Adobe Integrated Runtime application). This object oriented language is more adherent to the ECMA-262 standard which many other languages (i.e. JavaScript) follow.

This class gives non-programmers a chance to really understand the benefits of using a programming language in an animation tool like Adobe Flash. Students will learn the basics of the core language of ActionScript 3.0, how it is used within the timeline, interact with objects (movieclips) on the stage and library, develop simple animations with just ActionScript and the importance and how to create preloaders for your movies.

This beginner class on ActionScript 3.0 will give you a great foundation for creating (or understanding existing scripts) your own custom scripts.

## Audience for this Class

This is a beginner class on ActionScript 3.0 and I assume no previous programming experience.

This class is for anybody who (pick one):

1. A good understanding of the fundamentals of Adobe Flash (using movie clips)
2. Have taken the [Flash Introduction](#) and [Flash Intermediate](#) classes at LVSONline.com
3. Have previous experience with ActionScript (version 1 or 2).

## Tools Required

You need to have **Adobe Flash CS3 or CS4** or newer. If you have Adobe CS3, you will need to notify me so I can email you the examples directly. All weekly zip files will be in the Flash CS4 format.

## Following the evolution of ActionScript 3.0

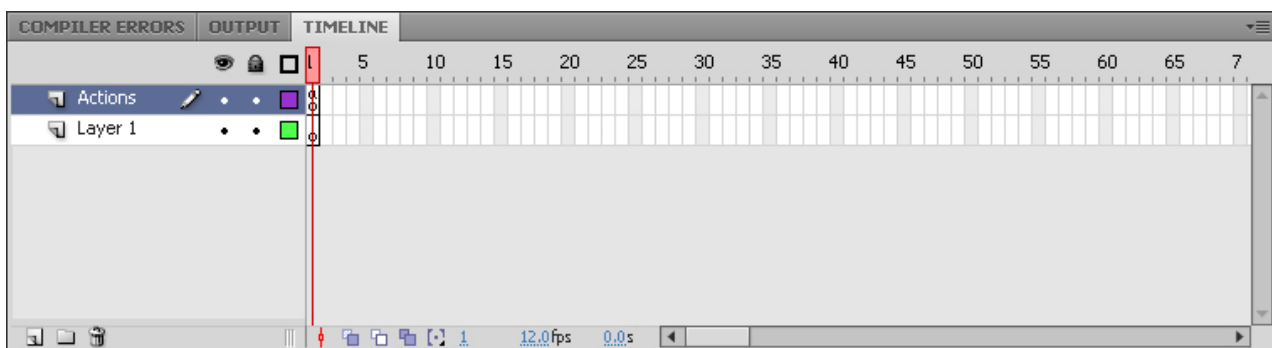
Elements of Flash-based scripts can be traced back as early as Flash 2. However, the name “ActionScript” did not make an appearance until the release of Flash 5 in 2000 with the release of ActionScript 1.0. ActionScript was originally a scripting language built to aid in the navigation of the Flash animation environment. These simple scripts were nothing more than the ability to change frames or scenes. With each new release of Flash, however, ActionScript becomes more and more adherent to the ECMA-262 standard, which allows for an even greater degree of optimization.

**Note:** [Ecma International](#) (formerly the European Computer Manufacturers Association) is an organization responsible for the standardization of information technology and communication. The ECMA-262 standard, also known as [ECMAScript](#), is typically associated with the standardization of many popular web dialects such as JavaScript and ActionScript.

This version offers an ActionScript language that has been completely reconstructed from the ground up. Though much of the base syntax remains, it is often said that ActionScript 3.0 should be approached as an entirely new language, independent of previous releases of ActionScript.

The Flash-based release of ActionScript 3.0 saw many key architectural improvements such as a true object-oriented model, enhanced low-level access, and a revamped version of the ActionScript Virtual Machine (AVM2). All of these improvements combined to create blistering fast performance and a greater degree of optimization.

## Workspace (Window -> Workspace -> Developer)



Personal preference determines how you want your Flash environment to be laid out, but be aware that since we will be more focused on the Actions Panel (F9), you may want to try out the Developer Workspace to see how you like it.

Since most students will be coming into this class as designer/animators and the Timeline was your normal primary focus this may feel a little uncomfortable at first (especially since the Timeline is not visible by default), but you may find you like this layout in time.

**Note:** You may notice in the Timeline the little **a** above the empty keyframe symbol (empty circle) in a frame. This identifies that that frame contains ActionScript code and is the only place in Flash (apart from an external .as file) that you can put it. Also, you may notice that I put this in a separate layer (Actions); this is the standard (but not required) way of identifying exactly where your ActionScript code will be. It just makes it easier to quickly find your scripts when your Flash movie contains many frames, folders, etc.

## **Crafting your first ActionScript application**

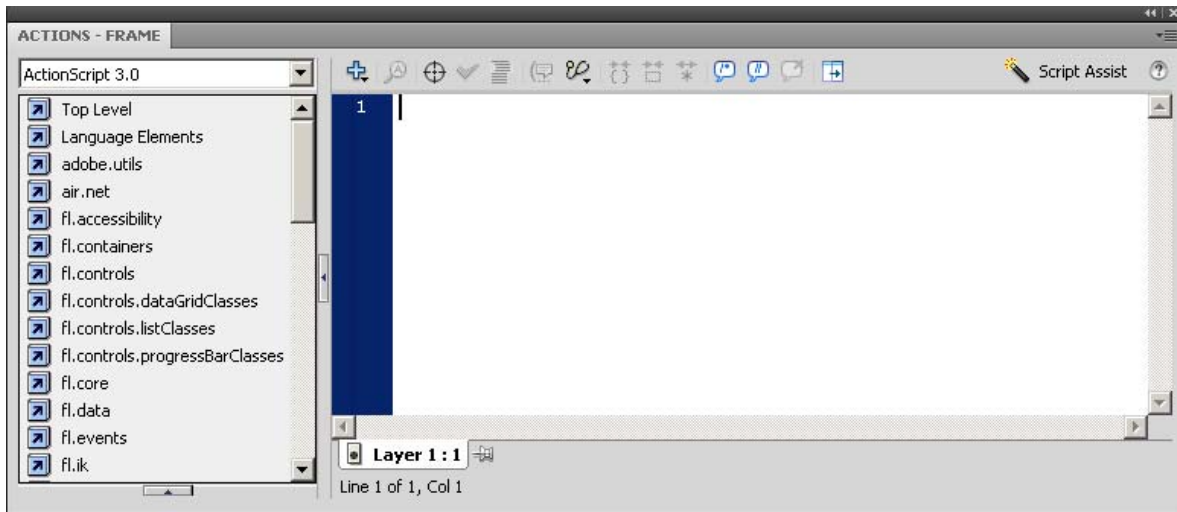
In the course of computer programming history, there have been many great traditions passed down from programmer to programmer. One of the most famous practices of ritualistic behavior comes in the form of the [Hello World](#) application.

The Hello World application is typically used as the very first example in a book or as the first computer program written by a new programmer. It is basically nothing more than the words “Hello World” being displayed on the screen.

The exercise will then serve as your frame of reference. So without further ado, it's time for your first program to say “Hello!”

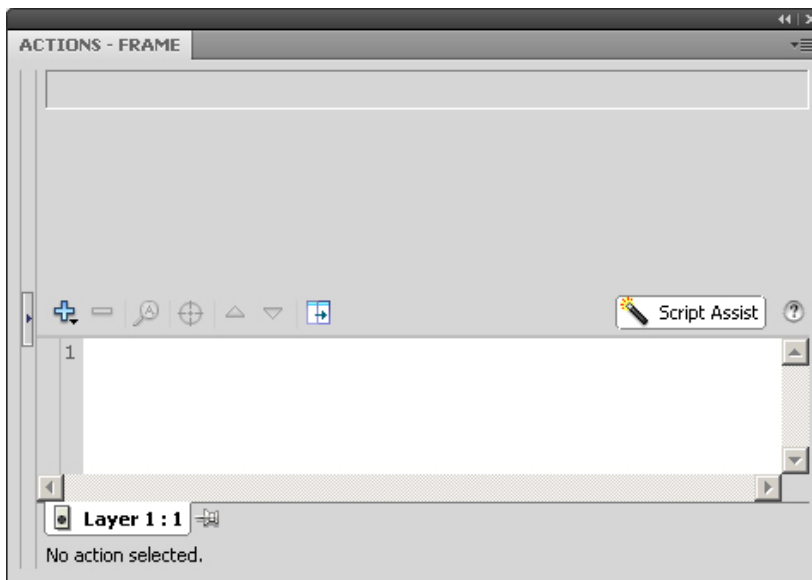
1. Open Flash CS4.
2. Create a New Flash file (ActionScript 3.0) by either choosing the Flash File (ActionScript 3.0) option on the Flash Welcome Screen or selecting File -> New and choosing Flash File (ActionScript 3.0) from the New Document window.
3. Once your new document has been created, open the Actions panel by selecting

Window -> Actions or by pressing F9. (Figure 1)



**Figure 1 – Actions Panel**

4. Once the Actions panel is open, activate the Script Assist by clicking the Script Assist button, located in the upper right of the Actions panel. At this point you will notice the expansion of the Script Assist above the script pane. You should also notice that not much is happening there at the moment (Figure 2).



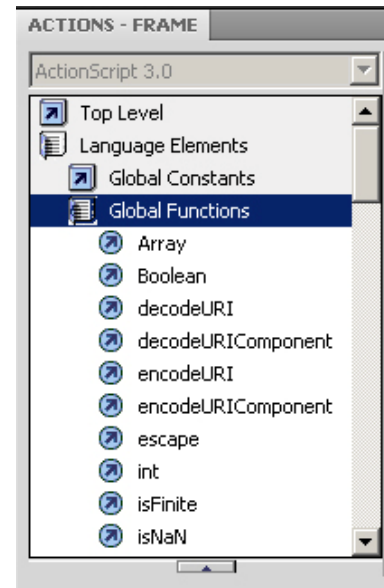
**Figure 2 – Script Assist Activated in Actions Panel**

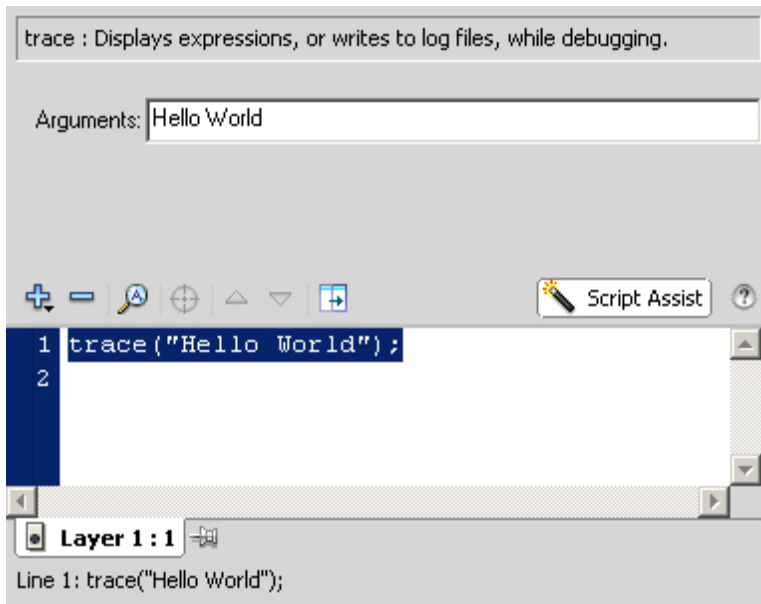
5. In order to use the Script Assist, you will need to add a code snippet from the Actions toolbox. The Actions toolbox is located in the upper right of the Actions panel. Make sure that the scrollbar in the Actions toolbox is at the topmost position. Then select Language Elements, followed by Global Functions.

6. One of the coolest features in the Script Assist is the fact that you can see what each function does before you use it. With Global Functions expanded, scroll down until you see the trace function. Single-click the trace function to see what it is used for.

7. Add a trace statement to your ActionScript by either double-clicking it or dragging it to the script pane. Immediately, your script is added to the script pane and a list of parameters is now visible in the Script Assist pane.

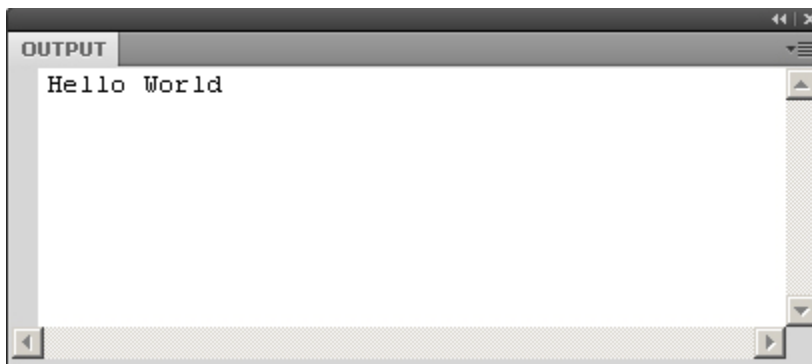
8. Now that the function has been added to your ActionScript, you will need to fill out the Arguments field to get it to work properly. As you will find out shortly, the trace statement is primarily used to write information to the Output panel in Flash. For now it will be sufficient for you to type “Hello World!” (including the quotation marks) as shown in Figure 3.





**Figure 3 – Adding a trace statement with “Hello World” argument**

9. Test your movie by selecting Control -> Test Movie or by pressing Ctrl-Enter/Option- Enter. This will publish your SWF file, and immediately you will see the Output panel appear with the phrase “Hello World!,” as shown in Figure 4.



**Figure 4 – Output window with Hello World**

Congratulations! You have just written your first ActionScript program

## The trace statement: Leaving breadcrumbs

The trace statement is actually a special kind of statement known as a top-level function.

The primary function of the trace statement is to send any expression to the Output panel during author-time testing of your Flash movies. You may also send output to a log file during debugging. Typically, trace statements are used to check the execution timing and values of dynamic parts of any given program.

With the trace statement you also have the ability to print multiple items to the screen at one time. To do this, simply type the values you would like to trace delimited by a comma. With the following example, the characters a, &, and b would all print to the Output panel.

```
trace("a" + "&" + "b"); // a&b
```

The trace statement is one of the most useful allies that any ActionScript developer can have. It will allow you to communicate with yourself as a program executes.

## Script Assist—taking it easy

The true benefit of this tool is that it allows developers to work in tandem with the Actions toolbox to quickly develop scripts that are meaningful to their program.

By single-clicking items in the toolbox, the Script Assist will display a brief description of that item. Double-clicking an item will then add it to the script pane.

Once an item has been added to the script pane, the Script Assist then displays a list of fields that are relevant to properly constructing that section of code. Therefore, a beginner can quickly create well-formed ActionScript with a just few clicks and filling out a couple of fields. Because the Script Assist is so user friendly, it is often an excellent starting point for someone wanting to learn ActionScript without the worry or hassle of proper syntax and formatting.

**Note:** The Script Assist is meant for simple scripting. Therefore, it is only available through the Actions panel while creating embedded code.

## Basic elements of ActionScript programming

Most programming languages within a certain categorization operate in pretty much the same fashion. By “categorization,” we mean the type of language. Specifically we are talking about [object-oriented languages](#).

Even though this class is focused to the ActionScript language, the programming basics that you will learn here are easily transferable to many other object-oriented languages.

Just like any written language, programming languages are also governed by a specific set of rules that need to be followed in order for the statements to make sense to the computer. Fortunately, this set of rules is infinitely less complicated than those applied to written language.

### Case sensitivity

ActionScript is classified as a case-sensitive language. The following example shows the declaration of two variables, `mylist` and `myList`. At first glance they may look the same, but the capitalization of the letter “L” in the second variable name is enough differentiation for these to be treated as completely different elements.

```
var myList:String;  
var mylist:String;
```

Though this may not seem like a tremendous issue, it is the small details like this that wreak havoc on many programs. As a new programmer, case sensitivity will more than likely be the culprit for the majority of your programs not working properly.

**Note:** In Flash development it has become commonplace for Flash programmers to use a typing technique called [camel casing](#), which is applied to an element that is given a name comprised of several words not separated by spaces.

### Dot syntax

In ActionScript, dot notation is used to perform two primary functions. First, you can use the dot operator to import libraries into your Flash program. As you begin to break a program into manageable chunks, you will want to begin to create ActionScript files

externally. A library is nothing more than a collection of external ActionScript files that is stored in a centralized location.

The second use for the dot operator is to access the members of a particular object. Again, don't be confused by the phraseology. As you will learn very shortly, objects used in object oriented programming are comprised of properties, methods, and events. These items are known collectively as the members of the object. And, the dot operator will grant you access to some of these members for the benefit of your program.

For sake of simplicity, we will pretend that I (Frank) am an ActionScript object called Frank. As an object I could contain various properties like height, weight, and whether I have hair. Similarly, I could have a method for performing various tasks like eating, sleeping, and working.

Now if we wanted to access one of those properties or tasks, we would do so using dot notation. The following example shows access to various members of the Frank object:

```
Frank.height = 6.00;  
Frank.weight = 185;  
Frank.hair = true;  
Frank.hair.color = "brown";
```

```
Frank.eating();  
Frank.sleeping();  
Frank.walking();
```

## Expressions and literals

Literals refer to values in a program that are typed (keyed in) and returned verbatim. In the following example the variable `myName` is assigned the literal value of "Frank". When the `trace` statement is used to print this variable to the Output panel in Flash, you will notice that it returns the value Paul exactly as it was assigned.

```
var myName:String = "Frank"; //String literal  
var currentAge:int = 5; //Numeric literal  
var birthday:int = currentAge + 1; //Numeric expression  
trace(myName + " is " + birthday + " years old.");
```

Conversely, expressions are values that are resolved by the execution of a statement. The variable `birthday` in itself does not have a legitimate value. It is dependant on the value of the variable `currentAge`. Therefore, the expression `currentAge + 1` must resolve to a legitimate value before any value can be assigned to `birthday`.

If you were to delete the second line of the preceding program, you would notice that the expression would be unable to resolve, and an error would appear. Fortunately, in this case, the value does resolve to 6.

Finally, all of the items in the parentheses of the trace statement would also be considered an expression. Though there are several string literals present, the entire group of elements needs to resolve to a single value before it can be passed to the Output panel.

## Semicolons

In Flash the semicolon is used to indicate the end of an executable statement. You should think of this as a period in your program. Though semicolons are not required, not using them could cause some unexpected results while programming. For example, omitting a hard return between two lines of code that are also not delimited by a semicolon would cause your program to fail to execute. Further, it is considered good practice to use them.

The following two code examples look almost identical to one another. The first example is missing the semicolon after the string `Frank`. Without this semicolon, ActionScript cannot tell that each line actually contains two statements. Therefore, an error occurs in the first line.

```
var myName:String = "Frank"      trace(myName); // error
var myName:String = "Frank";    trace(myName); // frank
```

## Comments

Comments are an extremely important part of any program. Though you may only be writing a program for your own benefit, chances are you are actually writing the program for the benefit of a team or company. Regardless of the end result, it is more than likely that someone else will eventually have to go into your code and make some kind of tweak or edit.

It is considered excellent practice and extremely courteous to properly comment your code.

Commenting allows programmers to type additional text among their program's code offering directions, instruction, or additional insights. Any line that is commented will be ignored by the compiler and will neither interact nor interfere with any part of your program. In Flash, there are two ways to generate comments. For a single-line comment, a programmer can use double (//) slashes. All characters to the right of the double slash will be commented out.

The second method of commenting is the multi-line comment. This is achieved by using the single slash and asterisk (/\*) character combination to open the comment and the asterisk and single slash (\*/) character combination to close the comment.

## Variables

Variables are the most basic component of any given computer program. Technically speaking, a variable is a reference to a portion of memory that has been allocated for the storing of a particular type of data. Basically, this is a fancy way of saying it is a name given to a location where a specific kind of information will be stored. And a computer program is nothing more than a sophisticated way of manipulating information. In some capacity or another, every statement in a computer program must interact with a variable.

Therefore, variables can be thought of as the subject or noun of the computer program. To declare a variable in ActionScript, you must first use the `var` keyword.

For example, the following statement declares the variable `myVariable`. Failing to use the `var` keyword in the declaration of a variable will result in an error in your program.

```
var myVariable;
```

As a programmer you have complete control over the names you give your variables. However, there are a few rules that need to be followed when creating a variable name:

- Variable names can contain numbers or letters, dollar signs, and underscores.

- Variables names cannot begin with a number.
- They cannot contain spaces.
- Variable names must be unique. Two variables cannot share the same name within the same scope.
- Variable names are case sensitive. It is also recommended that you avoid using the same variable names with different case. For example, `myvar` is different from `myVar` but will probably create confusion.

Though it is not absolutely necessary, it is considered best practice to strictly type your variable by assigning it a data type. To strictly type a variable, add a colon (:) followed by the desired data type.

Once a variable is declared, you can give it a value by using an assignment operator (=) followed by the value. As shown here, our previously defined variables are assigned the values of Hello and 4:

```
var myVariable:String;  
var myNumber:Number;  
  
myVariable = "Hello";  
myNumber = 4;
```

You can also assign the value to a variable when it is created as follows:

```
var myVariable:String = "Hello";
```

Though it is also possible to instantiate multiple variables at one time using the comma delimiter, it is not considered best practice.

## Data types

If variables are thought of as the nouns of computer programming, data types can be thought of as the adjectives. A data type is used to describe what type of information is going to be stored in a variable. Though strict data typing is not required, it is considered excellent practice.

**Note:** It is important to understand that regardless of whether you define the data type for a variable or not, Flash will. If the variable is in use, it has a data type

Practically speaking, data typing can serve several purposes. First, strict data typing reduces the amount of memory needed for using any given variable. Therefore, if you data type your variables, the variable will only accept information of a specific type.

For instance, if a variable is typed String, you know the variable is going to be of the String data type and only store characters. Further, if a variable is type Number, you know that the variable will only accept numbers. Subsequently, if a variable is type String and you try to assign a value to it that is a number, you will receive a type-mismatch error from the compiler.

Second, typing your variables enables inline code hinting. Inline code hinting is a feature of the Actions panel whereby suggestions are made from the Actions panel as to what code should come next.

In ActionScript, data types can be classified into two categories:

- **Primitive:** Primitive data types are what we have been talking about so far. They include the most basic type of data that can be used in Flash programming
- **Complex:** Complex data types are every other type of data used in Flash. They include common reference data types such as Array, Date, and Math. Where primitive data type can only contain primitive types of data such as numbers and letters, complex data types can contain many primitive values and other complex values at the same time.

| Data type | Example       | Description  |
|-----------|---------------|--|
| Boolean   | True/False    | Values of this type can only be true or false. These are commonly used for comparison and decision making. |
| String    | "Hello World" | This type is used for any text-based value or string of characters.  |
| Number    | 1, 88, 4.3    | This type is used for any numerical value including floating-point or decimal values.                      |
| int       | 0, -5, 3      | This type is used for any integer or whole number.   |
| uint      | 1, 2, 3 . . . | Short for unsigned integer, this type can contain any whole number that is not negative or a decimal.      |
| void      |               | This type is used if a function does not return any value.   |
| *         | untyped       | This type is used if a variable is not of a specified type.  |
| undefined | undefined     | This type indicates untyped variables that have not been initialized.                                      |
| null      | null          | This type is used for variables that do not have a value at all.   |

**Figure 5 – Primitive data types**

## Operators

Adobe defines operators as special functions that take one or more operands and return a value. An operator, though defined as a function, is usually nothing more than one, two, or three characters used to take two or more values and evaluate them.

Keep in mind that because strictly typed variables can only accept the data of one type, the data type will also need to be changed from int to String for this to properly execute. In this case, the resulting join of two character strings is known as [concatenation](#).

```
var answer:String;
answer = "two" + "two";

trace(answer); // twotwo
```

## Postfix operators

Postfix operators are typically used to increment and decrement one numerical operand by 1.

As shown in the following example, the variable `a` is incremented using the increment (`++`) operator. It is then decreased using the decrement (`--`) operator.

```
var a:int = 0;
a++;
trace(a); // 1
a--;
trace(a); // 0
```

## Relational operators

Relational operators are used to compare the value of two operands. The resulting value is Boolean, either true or false. The following sample checks to see whether the value of the variable `a` is greater than or equal to the value of the expression `(1+2)`:

```
var a:int = 2;
trace(a >= (1+2)); //false
```

| Operator | Name                     | Description   |
|----------|--------------------------|---|
| <        | Less than                | Checks whether the left value is less than the right value                |
| >        | Greater than             | Checks whether the left value is greater than the right value             |
| <=       | Less than or equal to    | Checks whether the left value is less than or equal to the right value    |
| >=       | Greater than or equal to | Checks whether the left value is greater than or equal to the right value |

**Figure 6 – Standard relational operators**

## Equality operators

Equality operators work in much the same fashion as the relational operators in that they compare two values and return a Boolean value of either true or false.

In the next example the assignment operator (=) is used to give a value to the variable a. The first statement can be read a is equal to 2. Conversely, the trace statement uses the equality operator (==) to compare the values of a and the expression (1+2). Therefore, relational and equality operators can be thought of in terms of questions.

You can also use the NOT operator (!) to determine whether values are not related.

```
var a:int = 2;  
trace(a == (1+2)); //false  
trace(a != (1+2)); //true
```

| Operator | Name              | Description  |
|----------|-------------------|--|
| ==       | Equality          | Checks whether the left value is equal to the right value.   |
| !=       | Inequality        | Checks whether the left value is not equal to the right value.   |
| ===      | Strict equality   | Checks for same values, as well as compares the data types of each value. If the left value and the right value are the same and the data types are the same, the expression returns true. Objects and arrays are compared by reference, not data type.      |
| !==      | Strict inequality | Checks for the same values, as well as compares the data type of each value. If the left value and the right value are not equal or the data types are different, the expression returns false. Objects and arrays are compared by reference, not data type. |

**Figure 7 – Most common operators of equality**

## Logical operators

The logical operators are also similar to the relational and equality operators in that they compare the values of two operands. The primary difference is they give programmers the ability to compare multiple comparative statements. The following sample checks to see whether the value of `a` is greater than 1 and less than 3:

```
var a:int = 2;
if(a > 1 && a < 3)
{
    trace("Yes"); //Yes
}
```

| Operator | Name        | Description  |
|----------|-------------|--|
| &&       | Logical AND | Allows you to perform a comparison on one or more expressions simultaneously.  |
|          | Logical OR  | Allows you to perform a comparison of several expressions simultaneously. Only one of the expressions needs to be true for the statement to execute. |

**Figure 8 – The AND and OR logical operators**

## Conditional statements

Conditional statements are one of the first logical needs in any programming language. Quite simply they allow a programmer, or more to the point the program, to make an intelligent decision based on a set of predetermined conditions. For instance, if it is raining outside, wear a raincoat or else you'll get soaked.

## if . . . else statement

The `if...statement` is the simplest and most commonly used conditional statement in programming. It can be thought of as the fork-in-the-road decision maker. The statement is comprised of four primary parts. The `if` keyword simply lets the program know that it is going to be entering the `if` statement. The second part, characterized by parentheses, is where the actual decision is made. The third part consists of two curly braces that signify a code block associated with the `if` statement. Finally, all statements within the curly braces are executed if the `if` statement evaluates to `true`.

```
if (weather == "rain")
{
    putOnRaincoat();
}
```

The `if` statement works by evaluating expressions that are encapsulated within these parentheses. There are only two possible outcomes for the evaluation of any given expression with respect to an `if` statement, `true` or `false`. Therefore, the preceding example asks, “Does the value of the variable `weather` equal `rain`?” Again, the outcome can only be `true` or `false`.

The `else` clause can be added to the end of an `if` statement to offer a desired outcome for the `if` statement evaluating `false`. Therefore, rather than having your program do nothing, you have the ability to have it act intelligently with respect to either decision. As shown in the following code, the `else` clause enables the `if` statement to have an alternative option in the event it evaluates to `false`:

```
if (weather == "rain")
{
    putOnRaincoat();
}
else
{
    putOnShades();
}
```

## else . . . if clause

A third option for working with if statements is the use of the else...if clause. The following example shows how this option gives you the ability to break your decision making into multiple branches.

```
if (weather == "rain")
{
    putOnRaincoat();
}
else if (weather == "snow")
{
    putOnBoots();
}
else
{
    putOnShades();
}
```

## Logical operators && and ||

Finally, by using the logical operators && and ||, you have the ability to create compound evaluations to check multiple conditions at one time. The following sample code shows the use of both types of logical operators to evaluate compound conditions:

```
if ((weather == "rain") && (temperature == "4 degrees"))
{
    stayHome();
}

if ((weather == "snow" ) || (temperature == "4 degrees"))
{
    dressWarm();
}
```

## switch

The switch statement is a special kind of conditional that allows you to define a multitude of outcomes based on the evaluation of a single statement. Unlike the if statements,

which check only whether an expression is true or false, the switch statement checks the actual value of the variable, compares it to the list of viable options, and determines the appropriate code block to execute.

As shown next, the switch statement is defined by the switch keyword followed by a set of parentheses that contain the expression to be evaluated. All execution options are then encapsulated within the curly braces. Each subsequent option is defined by the case keyword, followed by an option value and a colon.

```
switch (weather)
{
    case "rain":
        putOnRaincoat();
        break;
    case "blizzard":
        putOnBoots();
        break;
    default:
        checkWeather();
        break;
}
```

## Loops

In addition to making decisions, it is also very common for a computer program to repeatedly execute a series of statements until a certain parameter is met. Loops are essentially statements that increment a variable a given number of times until a condition is met. In ActionScript the two most commonly used loops are the `for` and `while` loops.

### for

The most common loop used in programming languages like ActionScript is the `for` loop. The anatomy of a `for` loop is rather unique in that unlike other functions it uses the semicolon as the delimiter instead of the comma.

The reason for this is that you are actually sending three statements to the loop as opposed to an expressed value.

```

var i:int; //i is classically used for the incrementing variables

for (i = 0; i < 5; i++)
{
    trace(i); // 0 1 2 3 4
}

for (var i:int = 0; i < 5; i++)
{
    trace(i); // 0 1 2 3 4
}

```

The first of these statements, `i = 0`, sets the starting value for our count. The second statement sets the ending value of our count to 4. Finally, the third statement uses the incrementing postfix operator to increment the value of `i` by 1. These statements can be read as “For `i` is equal to 0 and `i` is less than 5, add 1 to `i`.”

The loop works as follows. With the first pass through the `for` loop, the value of `i` is 0. Therefore, the `trace` statement traces 0. When the loop has completed its first pass, `i` is incremented to 1. The loop compares this to the second statement. Is `i` less than 5? Yes!

The loop runs again. The `trace` statement traces 1 and the cycle repeats until the variable has reached the value as predetermined in the second statement—in this case 4.

## while

The `while` loop works in exactly the same manner as the `for` loop. As shown next, the first step is to define a base starting point for the incrementing variable. In this case, `i` will once again begin at 0.

```

var i:int = 0;

while (i < 5)
{
    trace(i); // 0 1 2 3 4
    i++; // Adds 1 to the current value of i
}

```

The `while` loop is a bit less complex than the `for` loop in that you now only need to give the `while` statement one conditional expression in the parentheses. Therefore, we again want this loop to run until it is less than 5, or 4. Finally, all code that we wish to have execute is placed in between the curly braces of the while function. As you can see, it is here that we tell our `i` variable to increment.

## Functions

Functions are the part of the program that makes things happen.

Think about the `trace` statement, which should now be very familiar to you. As you learned earlier, the `trace` statement is a special kind of function that passes information to the Output panel. This information comes in the form of a variable that can be either a literal or an expression. Like the `trace` statement, other functions have the ability to accept variable information in the form of arguments, also known as parameters, through the use of parentheses.

## Defining your own functions

Functions are defined using a special predefined keyword, `function`. This keyword works in the same manner as does the `var` keyword. Once a function is declared by using the `function` keyword, the function is then named under the same guidelines that govern the naming of variables. The function name is then always followed by a set of parentheses.

These parentheses are used to pass information to the inner workings of the function in the form of variables. These variables can also be declared in the parentheses at the time the function is declared. The function body is then established using a pair of curly braces. Within the function body, all statements that define the execution of the function are placed.

The following example shows the definition of a function called `helloWorld`. The `helloWorld` function accepts one parameter, `message`, of the `String` data type. The function will then pass the `message` variable value to the `trace` statement located in the function's body.

```
function helloWorld(message:String)
{
    trace(message);
}
```

In order to use a function, you simply need to type the function's name followed by the desired parameter value encapsulated in parentheses. The following example demonstrates how the `helloWorld` function can be used within the program. By passing "Hello World!" as the parameter, the `helloWorld` function will then trace the parameter to the Output panel.

```
function helloWorld(message:String)
{
    trace(message);
}

helloWorld("Hello World"); // Hello World!
```

**Note:** The term "call" is often used to describe when a function is used in a program. It is often said that you can "call a function" or "make a function call."

## Returning values from functions

For the time being we have been working with `trace` statements. The problem with the `trace` statement is that it always works. You put something in, you get something out. Given any logical parameter, the `trace` statement will give you some kind of meaningful feedback. Unfortunately, that is not indicative of how functions really work.

Functions can be thought of as a machine that operates on data. However, you don't always have to give it information. Additionally, it doesn't have to give you information back. In some cases, it will not want anything from you or give anything back. The `trace` statement, of course, is an elementary example of the other extreme.

You have learned how to pass information into a function. Remember, the parameters of functions that are defined in the parentheses allow you to pass information to the function.

Now let's take a look at how to get things back. The following function performs a basic arithmetic process on a couple of variables:

```
function mathMachine (a:int, b:int)
{
    var c:int;
    c = a + b;
}

trace(mathMachine(1,2)) ; // undefined
```

You will notice that when we try to trace the value of the `mathMachine` function, it returns the value of `undefined` to the Output panel. This is because the function is not returning any value. It is accepting two parameters, `a` and `b`. It is also, without question, adding `a` and `b` together and assigning that value to `c`. Right now, it simply isn't returning a value.

In order for a function to return a value, we have to use a special keyword, `return`. The `return` keyword does exactly what it says in that it returns the variable it is assigned to. To use the `return` keyword, enter it as you would keywords like `var` or `function` followed by the variable name you would like returned.

```
function mathMachine (a:int, b:int):int
{
    var c:int;
    c = a + b;
    return c;
}

trace(mathMachine(1,2)) ; // 3
```

After adding our `return` statement, you will notice that the function returns a value of `3`, as expected. Did you also notice the addition of the `:int` data type? Yes, functions can be data typed as well. The purpose of data typing functions is to make the program aware of what type of value is going to be returned by the function. Again, it is not required but considered extremely good practice.

## Variable scope

Scope refers to what parts of a program's code have the ability to reference a variable. Scope is always assigned automatically based on where the variable is declared. In ActionScript variables are defined by two different types of scope, global and local.

## Global variables

A global variable is one that can be accessed by all parts of your code. Global variables are defined independently of functions—that is, they reside outside of the body of any function. In the following example the variable `myVar` is declared outside of the body of the function `global`. Therefore, it is accessible from any part of the program. As you can see, both trace statements are able to trace the value of the variable `myVar`, which is `Hello`.

```
var myVar:String = "Hello";

function global()
{
    trace(myVar); // Hello
}
global();

trace(myVar); // Hello
```

## Local variables

Local variables, on the other hand, exist only in a small portion of your program. Local variables are declared within a function's body. They are only accessible directly by the function itself. As shown next, the `myVar` variable is declared inside the function's body.

Therefore, when we try to trace the variable using a `trace` statement located outside of the function, we receive a compiler error.

```
function local()
{
    var myVar:String = "Hello";
}
local();

trace(myVar); // error 1120: Access of undefined property myVar
```

## Having a little class or a big one

The last stop in the development of an ActionScript program is going to be the construction of a class. A class is a collection of related properties (variables) and methods (functions) that are grouped together in one collection. If you think of classes in the same way you think of functions, they are a means by which you can group similar code into one well-organized package. The idea of grouping code to make it more efficient is referred to as modularity. To get an idea of how this is helpful, consider the following list:

- A variable is the most basic element in a computer program.
- Statements are used to manipulate and change the information stored in variables.
- A series of repetitive statements can be organized into functions for the sake of efficiency.
- For an even greater degree of organization, functions and variables can be grouped together into what is known as a class.

In this example, I am going to show you how to construct a simple one-class application so you gain familiarity with the basic structure of an ActionScript-based class.

**Note:** Don't worry, we will come back to this in a later lesson in more detail.

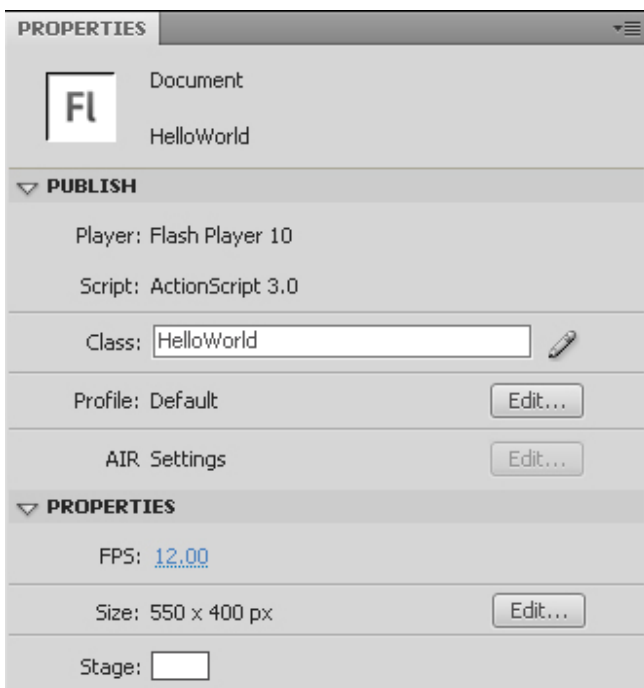
## Building your first application

First thing you need to do is open Flash, if you don't already have it open. Next, you need to create two new files. First, create a new Flash file by either selecting the Flash file (ActionScript 3.0) option from the Welcome Screen or by selecting File -> New and choosing Flash file (ActionScript 3.0) from the document window.

Save this file as `HelloWorld.fla` in a directory that you are comfortable with.

## The document class

The document class is a property of an FLA file that assigns any given class as the primary class to be used for this Flash file. Like all other document properties such as canvas size, background color, and frame rate, the document class can be set using the Property inspector.



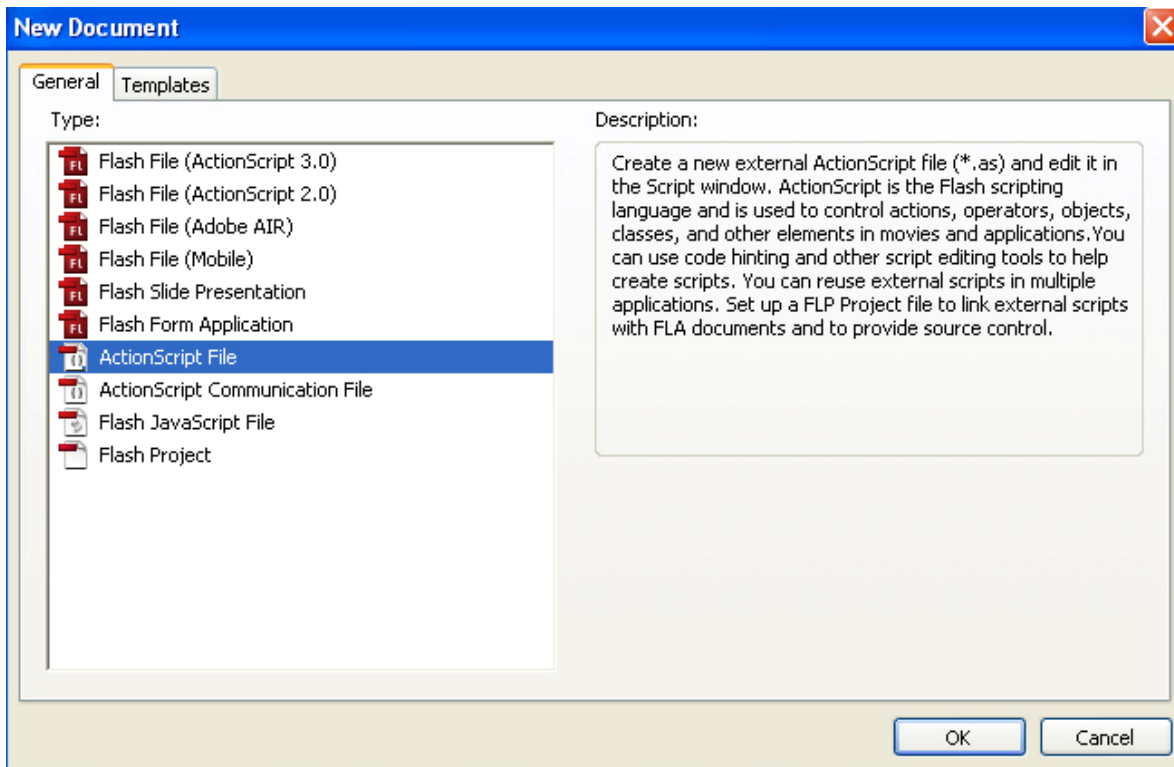
For this step we will need to have `HelloWorld.fla` as the active window in Flash.

To access the Property inspector, select Window -> Properties or press `Ctrl+F3`. To assign a document class, simply type the name of the ActionScript file you would like to use (without the `.as` extension) in the field labeled Class.

In the case of this example, you will be using the `HelloWorld` class. Therefore, type `HelloWorld` in the Class field of the Property inspector.

## Packages and import statements

Now that your files are properly set up, you can start defining your class file. You will need to create an ActionScript file by selecting File -> New and choosing ActionScript File from the document window. Save this as `HelloWorld.as` in the same directory as the `HelloWorld.fla` file.



**Figure 9 – Creating our HelloWorld.as (ActionScript file)**

**Note:** It is important that you save your ActionScript file in the same directory as your FLA; otherwise, Flash will be unable to locate it. We will later get into how to establish external libraries and define their locations.

The first step in defining a class is to properly define a package. In Flash, a package is nothing more than a collection of AS files. At this point, your AS file is located in the same directory as your FLA file, so it will not be necessary to give your package a name.

However, it is a required element of any custom class, so it will need to be added. Go ahead and add the following lines of code to HelloWorld.as:

```
package
{

}
```

The next thing that you will need to do is import other packages for use in Flash. Flash installs with core functionality that is comprised of hundreds of classes. In order to use any functionality from another class, you will need to import it. To do this, you will use the `import` statement followed by the location of the class. Because you will need to use some of the functionality from the `display` and `text` packages, you will need to import those.

```
package
{
    import flash.display.*;
    import flash.text.*;
}
```

The preceding example uses the `import` statement to import the classes from the `display` and `text` packages. In the preceding code, `flash` represents the physical location of those files on your computer's hard drive. `display` and `text` represent folders in that location.

These folders are what we referred to as packages. Inside each package is a varying number of AS files. The asterisk here represents "all" of the AS files in that package. Therefore, these two statements have imported all of the classes from both the `display` and `text` packages.

## Class definition

The next thing that you need to do is define your class. As you can see in the next example, you'll have to add quite a bit of text in the form of keywords. For now, all you really need to be concerned with are the words `class` and `HelloWorld`. The rest, though necessary, will be explained in more in a later lesson.

Declaring a class is actually the same process as declaring a variable or function. You need the reserved keyword, `class`, to let the compiler know that this is indeed a class. And you need an appropriate name. The naming of a class is extremely important. It must be the exact same name as the AS file that it resides in, case and all. Therefore, because this class is being written in the `HelloWorld.as` file, it needs to be called `HelloWorld`.

```
package
{
    import flash.display.*;
    import flash.text.*;

    public class HelloWorld extends MovieClip
    {

    }
}
```

## Constructor functions

Constructor functions are the last absolutely necessary piece of any given class. When a class is instantiated, or declared in the program, the constructor function is responsible for what happens. It is the initializer of the class's functionality. It is the first domino, so to speak.

Like any other function in ActionScript, the constructor is declared using the `function` keyword and named appropriately. As with the class, the naming of the constructor is crucial. It must be the same as the name of the class and the file it resides in. For this example, the constructor must be `HelloWorld`.

```

package
{
    import flash.display.*;
    import flash.text.*;

    public class HelloWorld extends MovieClip
    {
        public function HelloWorld()
        {

        }
    }
}

```

## Wrapping it up

To finish up your first class, you need only add the nuts and bolts to the constructor function. By now this should be a fairly simple task. What you are doing here is emulate the `trace` function. Because the `trace` function does not render anything to your published SWF file, you are going to need to fake it using a simple text field.

In the upcoming example, you will create the variable `helloText` and give it a data type of `TextField`. A `TextField` is considered a complex data type. Complex data types are named so because they can contain complex sets of data. This means that they can represent entire classes. Classes can contain members, which can consist of properties (variables) and methods (functions). Therefore, in direct contrast to a primitive data type like `int`, which is simply a whole number like 3, a complex data type can contain an abundance of information ranging from primitive data types or other complex data types.

When you assign a value to `helloText`, you will use the `new` keyword followed by the `TextField()` function. In this process, you are instantiating an object based on the `TextField` class. So, when you use the `new` keyword to assign a value to `helloText`, you are actually referencing the constructor function of the `TextField` class to create a new text field.

You also learned that an object, like a `TextField`, can have properties. These properties can be accessed using dot notation. In this case, text fields have a property called `text`,

which is nothing more than a variable that represents the text displayed in the text field. So, the second statement in your constructor function will essentially take a `TextField` object named `helloText` and set the value of its `text` property to `HelloWorld!`

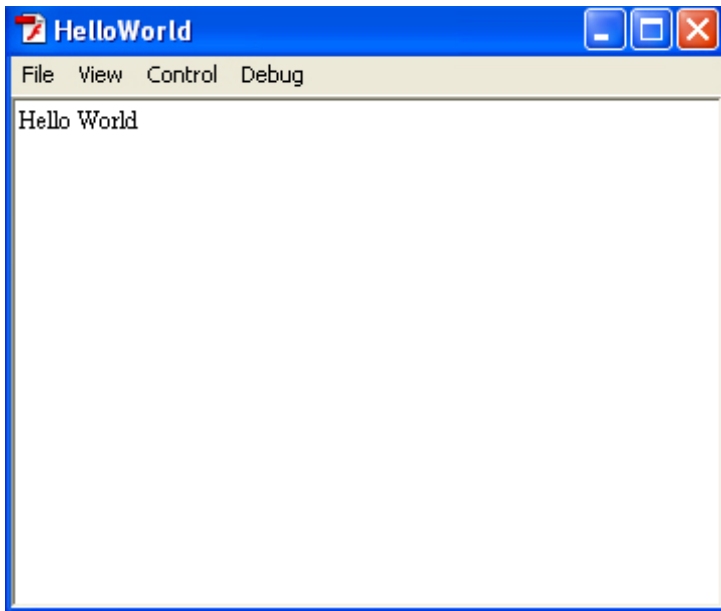
Finally, to get this `TextField` to display on the stage of your SWF, you need to add it to the display list. This is accomplished using the `addChild()` method. Therefore, you will add the `helloText` `TextField` to the stage with the statement `addChild(helloText)`.

The following shows the addition of the three previously mentioned statements to the `HelloWorld` constructor function:

```
package
{
    import flash.display.*;
    import flash.text.*;

    public class HelloWorld extends MovieClip
    {
        public function HelloWorld()
        {
            var helloText:TextField = new TextField();
            helloText.text = "Hello World";
            addChild(helloText);
        }
    }
}
```

Go ahead and save your `HelloWorld.as` file and press `Ctrl+Enter/Option+Enter` to publish your SWF. You should see the text “Hello World!” publish to the stage!



**Figure 10 – Publishing our movie (HelloWorld fla)**

### **Additional Resources:**

1. Adobe ActionScript 3.0 Reference:

[http://help.adobe.com/en\\_US/ActionScript/3.0\\_ProgrammingAS3/index.html](http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/index.html)

2. Flash CS4 ActionScript 3.0 Language Reference:

[http://help.adobe.com/en\\_US/AS3LCR/Flash\\_10.0/](http://help.adobe.com/en_US/AS3LCR/Flash_10.0/)

3. Adobe ActionScript Technology Center:

[http://www.adobe.com/devnet/actionscript/getting\\_started.html](http://www.adobe.com/devnet/actionscript/getting_started.html)

4. Getting Started with AS 3.0:

<http://tv.adobe.com/watch/actionscript-11-with-doug-winnie/working-with-the-actions-panel-episode-2>

## Assignment for Lesson 1

1. Read over this lesson (I know we covered a lot).
2. Review the examples (wk1.zip) and try out the syntax in creating little simple examples to get used to what we have covered in the first lesson.

**Note:** As I mentioned we covered a lot in this first lesson, but I wanted to get the basic core syntax out of the way so we can go into some fun stuff in the upcoming lesson(s).

If you are confused about the `HelloWorld` class example and how we created a separate AS file to store our class definition, don't worry about it. We will come back to that in a later lesson.

**Note:** You also may ask if you can use AS files to just store functions and/or variables instead of classes. Yes, you can. Just use the include statement.

```
include codefile.as
```

3. Post any questions or code snippets (not the SWF) you have created on the class discussion board:

<http://actionscript30-desingers.com/frank-lesson1.html>

I will respond with a review on the class discussion board.

Copyright 2010 © Frank Stepanski

Used with Permission: LVS Online Classes / LVS Associates

Lessons, files and content of these classes cannot be reproduced and/or published without the express written consent of the author.

Use of this site implies agreement with the [Terms of Use](#)