



By Lyndon Daniels

This work is licensed under a [Creative Commons Attribution-Noncommercial-Share Alike 3.0 License](https://creativecommons.org/licenses/by-nc-sa/3.0/).



You are free to share, modify and redistribute this work under the following conditions

- 1) Attribution must be given to the author.
 - 1a) All derivatives of this documentation must mention the author by name and include free access to the original documentation electronically.
 - 1b) All non-educational derivatives of the exercises (.FLA and .AS files) must reference the original files and author in comments in the source code, and include a location to freely access the original exercises (.FLA and .AS files).
- 2) No direct commercial profit, monetary or other, is to be gained from this documentation. This includes selling the original documentation or any derivatives.
- 3) All derivatives of this documentation are to be released under the same Creative Commons License that this documentation is released under.

copyright Lyndon Daniels 2008

Please note when copying and pasting code from this document into the actions panel in Flash, you will need to change the special word processor quotation marks “” to standard quotation marks "" . If left in their word processor form your compiler will throw an exception or error.

Preface

An Introduction to Programming with ActionScript pg2

Chapter 1

Standardized Coding Practices pg3

Chapter 2

Errors pg4

Chapter 3

Variables pg6

Chapter 4

Strings pg8

Chapter 5

Numbers pg9

Chapter 6

Controlling the Flow of Execution pg12

Appendix

Commonly Used Flash Commands pg16

Preface

An Introduction To Programming With ActionScript

Higher Level Languages

The oldest programming language is based on combinations of two numbers 0 and 1, this code is known as Binary and at some point computer hardware has to process information in this language. Fortunately we don't have to understand Binary in order to interact with a computer's hardware. Software, applications and operating systems take care of that for us. In Flash we simply deal with, what is known as, a higher level language. JavaScript and Python are also examples of higher level languages. Higher level languages are also sometimes referred to as compiled languages or interpreted languages. The term Higher level is used because the language of Flash, that is ActionScript, has moved away from Lower level interactions with hardware and all of those lower level interactions have been taken care of for us. For example when we tell an object in ActionScript to move across the stage, we don't also need to tell the computer's processor what to do, or tell the computer's screen how to redraw the object that's all taken care of for us. This lower level interaction in Flash's case happens through a language known as C. But C is still a far cry from Binary code!

Chapter 1

Standardized Coding Practices

Standardized coding practices are there to establish a consistency between different software languages. If you do not use them your software may or may not throw an exception, but you can be certain that there are many people that will! Use these practices and avoid creating your own, they have been tried and tested for many years.

Comments

Comments are lines of information inserted between code that informs a person reading the code what the purpose of the next line or next few lines of code is. Comments are completely ignored by the software executing the code. They should be written in plain and simple English or, what ever your chosen language, however consider that the comments you write might not always be for your personal understanding but for another person reading your code.

Comments are particularly useful for code that you have not used in several months. Regardless of whether you wrote the code or not, trying to understand what revisited code is supposed to do after long periods of time becomes a cumbersome process when it is not commented properly.

By the way it is estimated that 80% of a programmers time is spent maintaining code that already exists, this is largely why commenting code is a standardized coding practice.

White Space

Flash unlike some other software languages totally ignores white space. White space is the space between the characters that make up the lines of code for example a space, tab or enter/break. Just because Flash ignores white space doesn't mean you should. Using a consistent spacing and formatting of your code will make it more human readable. For example,

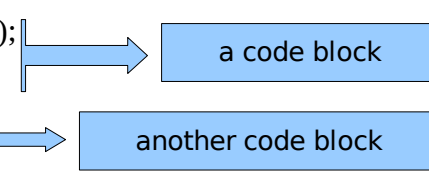
```
var firstNumber:Number=1;var nextNumber:Number=2;trace(firstNumber - nextNumber);
```

is a lot more difficult to read than this,

```
var firstNumber:Number=1;
var nextNumber:Number=2;
trace(firstNumber - nextNumber);
```

White space can be a particularly useful visual clue to separate code into task specific blocks. These blocks are referred to as code blocks.

```
if (1 == true) {
    trace("yep, 1 does indeed still mean true");
    trace("so true must also equal 1, right?");
} else {
    trace("somethings not right, here!");
};
```



Code blocks depicted with standardized coding practices will always have the same indentation per line of task specific code in Flash. One line will always follow the next, until the task of the code block is completed, at which point the indentation may change.

Chapter 2

Errors

Syntax errors

Computers process information in a very literal way. Although the meaning of a statement might be clear to a human reading the code, to a computer unless it is exactly literal it will be considered to be a “bug in the program”. For example the statement,

```
trase(“hello world!”);
```

Will throw an error, Even though any person reading this code knows that the person writing this code more than likely meant to type,

```
trace(“hello world!”);
```

Syntax errors are often caused by typo's and are generally easy to fix, once they have been identified.

Logical Errors

Logical errors are errors that do not cause the program to halt, crash or throw an error but will cause the program to act in an unexpected manner or produce unintended results. Logical errors can therefore be difficult to track down and rectify because the program (Flash) does not indicate the specific location of an error.

Common Mistakes resulting in Logical Errors

One of the most common mistakes amongst experienced and novice programmers is incorrect data typing. As noted earlier expressions are of certain types eg strings, numbers etc different data types should generally not be evaluated in the same expression using mathematical operators. For example a string cannot be subtracted, divided etc from or by a number “car”/12 = NaN. It does not make sense in the real world and nor does it make sense in programming. The following code is an example of a logical error in Flash,

```
firstNumber = "1";  
nextNumber = "2";  
trace (firstNumber + nextNumber);
```

One would expect the answer to evaluate to 3, but the answer Flash returns is 12. Flash is adding two expressions that have both been data typed as strings. The process of adding strings together is referred to as concatenation.

A better way of declaring variables is to data type them from the beginning. That way you can be certain of the type of data you are dealing with and if you try evaluating them in the same expression Flash will throw an error. An example of data typing variables at declaration follows

```
var firstNumber:Number = 1;  
var nextNumber:Number = 2;
```

```
trace (firstNumber + nextNumber);
```

The answer now evaluates to 3 as expected.

Just to make things a little more interesting, Flash supports a feature known as data type converting. This might seem like a contradiction to what I just said regarding “strict” data typing, but it can be quite useful. As you probably know data type converting does in fact allow you to use mathematical operators between numbers, certain strings and booleans. To summarize it is not necessary to data type variables but certainly a recommended practice. Here's a poor example of data type conversion,

```
var firstNumber:String = "1";  
var nextNumber:Number = 2;  
trace (firstNumber - nextNumber);
```

returns a value of -1. This is an example of data type conversion. A scenario like this could and should be avoided like so

```
var firstNumber:String = "1";  
var convertedNumber:Number = int(firstNumber);  
var nextNumber:Number = 2;  
trace (convertedNumber - nextNumber);
```

a new variable is created and data typed as a number to convert the string to an integer. The trace therefore is not data type converting, but in fact dealing with two numbers.

Chapter 3

Variables

Variables create a means of storing a value under a name and provide us, through this name, the ability to reference this value (which may be a simple number or a complex expression) and use this value in an expression.

Variables are created in the client computer's memory and are stored with their associated value. It is far easier to reference a value using its variable name than to know exactly where this value exists in a client computer's memory and refer to it as such.

Lets take a real world example to demonstrate how we use variables in programming, say you had a cell phone with your friend's name and number on it. Your friends name is "john", this is the variable name. His cell number is "+27829662141", this is the variable's value. If you were to call your friend on his cell phone you do not have to know where he is in order to get a hold of him. In the same sense you don't need to know where in your computer's memory a variable's value is stored, you just refer to the variable's name to get its value. As you can imagine this makes things a lot easier. In the same way referring to a value through its variable name will make scripting a lot easier especially when your variable has a constantly changing value assigned to it.

Variables, as the name implies do not have to be constant in their value. Their value can change at any point. What cannot change about a variable is the way you refer to it. A variable's name must always remain consistent, there is no way of changing a variable name however you can create a new variable with a different name and assign it the value of your previous variable via its name. To elaborate on the previous "real world" example of your friend and his cell phone number. The person you know as "john" will always be "john" however his cell phone number (particularly if he's south African) may change from time to time, that is to say his name (or the way you refer to your variable) stays the same but his number (or the value assigned to the variable) is prone to change.

Creating Variables

The process of creating a variable is generally referred to as declaring a variable. The best way to declare a variable in Flash is to assign your variable a name, data type it and if necessary assign a value to it. For example,

```
var friendName:String = 'john';
```

This is referred to as an assignment statement, as we are assigning the value that is on the right of the equals sign to whatever is on the left of the equals sign.

Using variables can save us from a lot of unnecessary typing. Variables can have any value assigned to them you like they can have a simple string like "john", a number or a complex expression assigned to them. Here's an example of how to use a variable within a trace statement,

```
trace("Hi, my name is" + friendName + "!");
```

Following from our last example Flash would print the following string in the Output window,

Hi, my name is john!

Creating Legal Variable Names

- Variable names can only contain alphanumeric characters and underscores.
- Variable names cannot start with numbers.
- Variable names should be descriptive but not verbose.
- Use consistent casing. Variable names generally do not start with an uppercase character unless the variable is assigned the value of an instantiated object.
- Your variable names should be self documenting, that is to say they should be descriptive about their purpose without having to include additional comments explaining their purpose.
- Variables that are typed in all uppercase characters are referred to as constants. Their value is not supposed to change. This is simply a naming convention and there is nothing stopping you from changing the value of a constant in Flash, although this is not a recommended workflow. So beware when using this naming convention as it could lead to confusion if not used properly.

Chapter 4

Strings

Strings are literal values that usually denote some sort of descriptive meaning using alphanumeric or punctuation characters. They can also contain several elements for example the string “John” is made up of a J, o, h and an n. Strings can also contain white space characters such as a space or a tab, eg “ is a string. This is also considered as an element in exactly the same way a “J” or an “o” for example is also considered a legal element of a string. When using strings in ActionScript the elements of that string must always be enclosed within quotation marks. The quotation marks must either be double or single, and always remain consistent that is to say that if you start a string with double quotation marks, you must end the string in the same way. Mixing double and single quotes will lead to errors. There is, however, an exception; if you wanted to include quotation marks as one or two of the elements making up your string you would start and end your quotation marks denoting a string value in the same way, then use the other quotation pair within the actual string. For example,

```
trace("John is the same as 'Jon'");
```

Concatenation of Strings

Concatenation is the process of linking together various things. In our case we are going to discuss concatenating strings. The concatenation operator for strings as you might have already guessed is exactly the same operator used in math, the plus sign “+”. to join several strings together we could use the operator like so,

```
trace("Hello" + " my name" + " " + "is Jon");
```

Notice this trace statement concatenates four strings together. It's not necessary to use as many strings as I did in this example but I think this code illustrates my point more clearly. This method of concatenating strings is okay, but a bit difficult to follow and not very flexible at all. A more commonly used technique would be to use variables data typed as strings then add the variables together. You will see this method of concatenating strings in code used more often by experienced programmers. That doesn't mean that if you don't consider yourself to be an experienced programmer that you should not use the technique. The example above is not very versatile if I wanted to repeat the statement it would mean having to include the whole line of code again. Although this might not seem like a major issue at this stage because the code is still very simple (even repeated), however when writing larger scripts repetitions code like this can make your code less human readable and also execute slower. It is therefore within your interests to consider changing and assigning values indirectly through variables for example,

```
var greeting:String = "Hello, my name is ";  
var name:String = "John";  
greeting = greeting + name;  
trace(greeting);
```

This code only uses two variables and is versatile in the sense that we could customize the name variable to any name we want and not have to worry about updating anything else in the script. Repeating the script is easy, as it's just a matter of typing a single word. Indirect manipulation of variables is something you will find invaluable when you start writing your own scripts.

Chapter 5

Numbers

Numbers in the programming sense work exactly as the numbers you're already familiar with. In fact programming is largely about the manipulation of numerical values. We get two main types of numbers in programming (although other languages may have more than two number data types)

- Integers
Integers are whole numbers, that is to say numbers without a decimal value eg 1, 5, 10, 789629386390, 0, -1, -25 etc are all integers
- Floating Point Numbers
Also known as floats are numbers with a fractional part that is they will always have a decimal point eg 1.9, 0.74653, -98668.1, 0.0 are all examples of floats.

Mathematical Operators

Mathematical operators are used to combine numbers to create new values eg +, -, *, /, % are the mathematical operators available in Flash, for example,

```
trace(2000 -1000 + 50);
```

I won't go into detail on the mathematical operators as I'm sure you are already very familiar with them, so in brief “+” plus, used for addition, “-” minus, used for subtraction, “*” times, used for multiplication, “/” divide, used for division. The final mathematical operator you might not be so familiar with. It is known as modulus and represented with this symbol “%”. Modulus is used to return the remainder of an evaluated expression. For example the command,

```
trace(107%4);
```

returns a value of 3, which is what is left over after 107 has been divided 26 times by 4

Number Types in Flash

As we already know there are two number data types in Flash, Integers and Floats. We have also touched on the topic of data type conversion. Flash treats all number data types as what it refers to simply as “numbers”. What this means is that flash will actually perform data type conversion between different number data types when necessary for example,

```
trace(12/5);
```

evaluates to 2.4. Now. If you've been paying attention you will notice that neither 12 nor 4 are floating point numbers, they are in fact both integers. Since the answer Flash has returned is a float we can assume that Flash has performed data type conversion from an integer to a float on both numbers. Dividing with floats is known as true division. If the answer was returned as an integer it would be 2, and that has got nothing to do with the program performing the calculation, rounding the value of 2.4 down because it does not. To arrive at the integer equivalent of 2.4 your software (Flash) would simply remove the decimal value so 2.4, 2.999, 2.00001 would all equate to 2 as an integer. The process of data type conversion in this case can be quite useful, but something you should be aware of none the less. This is also relevant because it's worth noting that floats on a

computer are usually rounded and therefore not exact. In Flash this is particularly relevant when your number's decimal value exceeds 17 digits.

Augmented Assignment Operators

Augmented assignment operators are used to assign a new value to a variable based on its original value, for example if we had the expression,

```
x = x + 5;
```

This is quite a commonly used expression and can become cumbersome to write over and over again. In augmented form this would be written as,

```
x += 5;
```

Other mathematical operators can be used in their augmented form in a similar way for example *= , += , -= , %= , /=

Two other augmented assignment operators exist. The long way of writing out the next two expressions follows,

```
x = x + 1;  
x = x - 1;
```

The augmented assignment operator versions of these two expressions consecutively read,

```
x++;  
x--;
```

As you can see using augmented assignment operators make your code easier to read, and can also contribute to make your code run faster.

Random Numbers

Random numbers are used to add an element of unpredictability to a program, for example they can be used to create a change in a computer opponents strategy, spawn enemies in a game at random locations, and create the basis of an AI opponents choices.

Random Numbers in Flash are generated between 0 and 1. Random numbers are generated using the random method of the Math Object. For example,

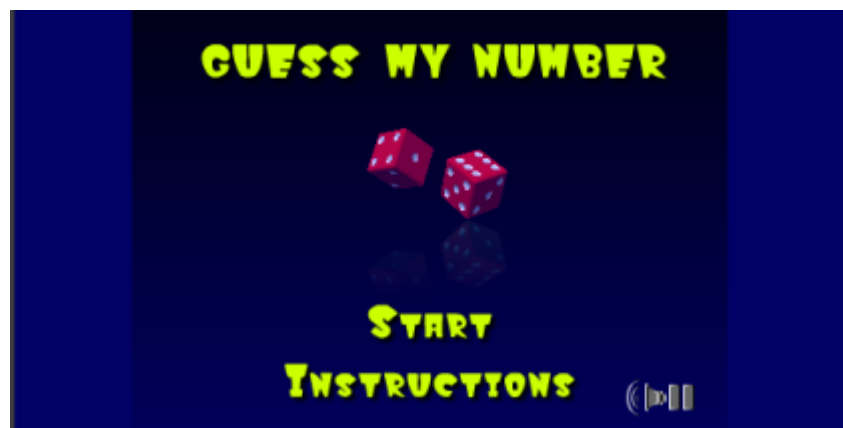
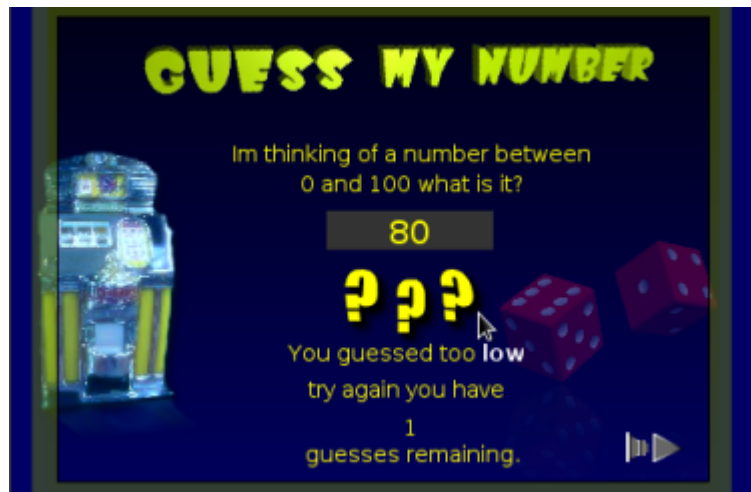
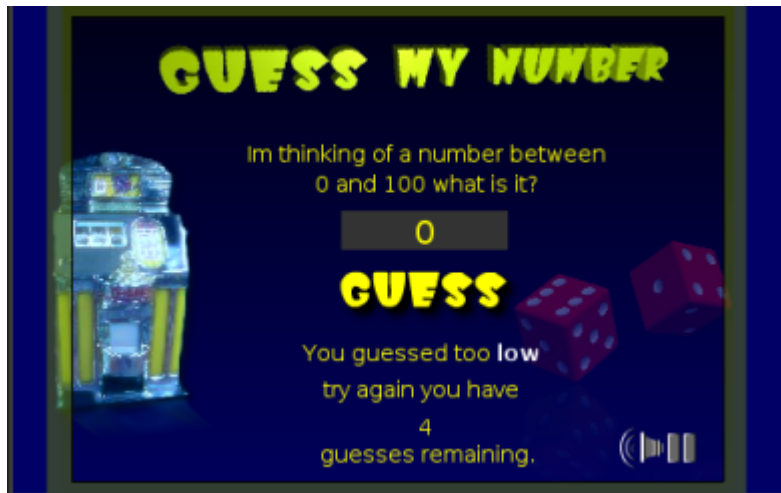
```
Math.random();
```

This will generate a random number and store it memory. This is not very useful because you cannot access the number if you need to use it in an expression. To generate a random number assign it to a variable then use it in an expression you would do something like this,

```
var randNum:Number = Math.random();  
trace(randNum);
```

It is, however, worth noting that although Flash refers to these numbers as random they are not sophisticatedly random enough to use in programming a secure site, so there goes that dream of an

online casino. If you want truly random numbers which are a lot more complicated and slower to create you would need to take into consideration factors such as radio active decay. However for the purposes of what we use Flash for, it is unlikely that you will need such sophisticated technology. See the Guess my number game for an example of random numbers.



Chapter 6

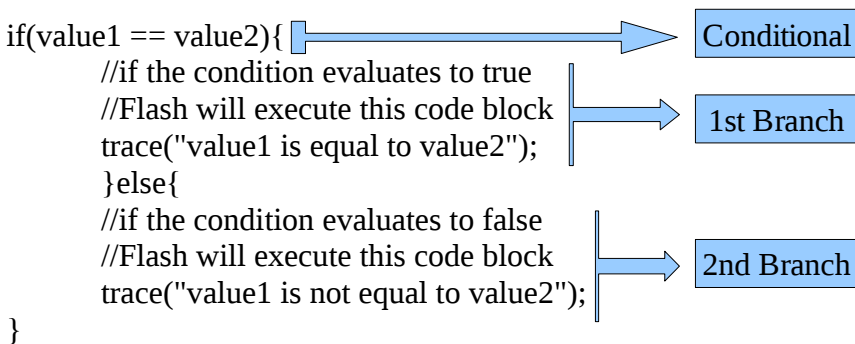
Controlling the Flow of Execution

When we write a script, generally we don't want the script to do the same thing every time we use it. Often what we want is a script that is versatile in its application. Excluding this versatility would make the program predictable and boring. We therefore use certain built-in commands that help us control the way in which our programs run, allowing us to skip blocks of code and execute other blocks of code before others in a non-linear way. This can be done in a dynamic and changing sequence based on the user's input. The process of creating this dynamic ordering (by allowing us to execute code in a non-linear order) is what is referred to as controlling the flow of execution.

The if Statement

Branching is the process of choosing one flow of execution over another based on the way the code is structured. The if statement allows us to create programs with the ability to branch off in different directions based on dynamic or predetermined values.

All if statements have a condition. A condition will either evaluate to true or false. The value of this condition determines which route your code will branch off into.



Comparisons and Comparison Operators

The conditions of an if statement are created by comparing values against each other. These values are compared using comparison operators. A list of Flash's comparison operators follows,

- == is equal to
- != is not equal to
- > is greater than
- < is less than
- >= is greater than or equal to
- <= is less than or equal to

Comparison operators work by comparing what is on their left side to that which is on their right side then returning a value of either true or false based on this comparison. For example;

```
if(5>4){  
    trace("true");  
}
```

The comparison in an if statement is enclosed between parenthesis. This above example is literally read in plain English as,

“If five is greater than four, trace true.”

What this simply means is that when this code is executed Flash will check if the value on the left side of the comparison operator is greater than that on the right of the operator. If the value on the left is greater than the value on the right (and as you are aware 5 is indeed greater than 4), Flash will return a value of “true”. Since the condition has evaluated to true flash will go ahead and execute the code in the curly brackets directly after the condition, in this case telling Flash to print the characters “true” in the output window. However if the condition evaluated to false flash would skip the code in the curly brackets and proceed to the next block of code following the closed curly bracket. Since there is no code following the closed curly bracket in this example Flash will simply evaluate the condition, realize that it equates to false and simply skip the trace command then do nothing, there after. Therefore by using a condition in our if statement we are controlling the flow of execution of our script and controlling what Flash eventually outputs to the user.

Please also note that the comparison operator == (is equal to) is not to be confused with the assignment operator = (equals). The former checks if the value on the left matches the value on the right of the operator, the latter assigns the value on the left, of the operator, to that which is on the right.

It's not only numbers that can be compared but expressions, variables and strings can also be compared. For example the if statement that follows compares two strings and returns a value of either true or false based on the strings position in the English dictionary

```
if("apple"<"orange"){  
    trace("apple comes before orange in the dictionary!");  
}
```

It is not common to make a comparison of this nature in scripting as you can see it's application is quite specific eg. perhaps the sort of thing that would be used in a spelling game. The condition has evaluated to true because “apple” is numerically less than “orange”, ie. comes before “orange” in the English dictionary.

The Structure of the if Statement

```
if (variable01==variable01){  
    trace("variable01 is equal to variable01");  
}
```

We have already seen how this script will execute, variable01 will be compared to itself and will obviously be the same, therefore return a result of true. This means the following code block between the curly brackets will execute. This is however pretty linear in terms of controlling the flow of execution. What if we could have our computer make a logical decision based on the condition provided then have two possible scenarios, one that is executed if the condition evaluated to true and another if the condition evaluated to false. We can with an if-else statement.

```
if (variable01==variable01){  
    trace("variable01 is equal to variable01");  
}else{  
    trace("somethings not right!");  
}
```

By adding the else clause to our if statement we have now provided our mini-application with two possible scenarios. If the condition evaluates to true the first block of code will be executed. If, however, the condition were to evaluate to false the first block of code between the first set of curly brackets would be ignored. Flash would then proceed to the else clause (if it exists, in this case it

does) and execute the code within the following set of curly braces (in this case telling Flash to print the text “somethings not right!” to the Output Window.

The if-else if statement

Using an if-else statement can be useful if you are only checking for one of two possible values. However what if you wanted to check for a specific value amongst many possible values. Using the else if clause you can check for as many different values as you like in the same if statement and provide an independent branch of code for each possible value. This is because each else if clause has it's own condition that can evaluate to true or false and Flash can subsequently choose which code block to execute. The else if clause is contained within the if statements curly brackets and it's syntax closely resembles that of the if statement itself.

```
if(variable01 == variable01){
    trace("variable01 is equal to variable01");
} else if (variable01 == variable02){
    trace("variable01 is equal to variable02");
} else if(variable01 == variable03){
    trace("variable01 is equal to variable02");
} else{
    trace("variable01 could be not be defined");
}
```

Adding else if clauses can be effective in capturing various possible values and useful for debugging a script.

The if-else if statement does not have to end with an else clause, but it can be useful to present a final set of instructions to the program in the case that none of the other clauses evaluate to true. It is possible for an if statement with several clauses to have more than one condition evaluate to true. In this case it is only the first condition, that is read from the top of the script down, that will have it's associated code block executed. All other conditions whether they evaluate to true or false will have their associated code blocks ignored. It is worth checking that your scripts do not fall into this trap. For an example of the else-if if statement check out the Guess my number Game.

Compound Conditions

Compound conditions allow you to have more than one condition within the same parenthesis in the same if statement's conditional and have your program make a decision based on a result accumulated from all conditions within the if statements conditional.

Compound conditions are combined with logical operators and involve two or more comparisons in the same condition.

The most commonly used logical operators are && and ||

The first is referred to as “and”.

All simple conditions within the same parenthesis, seperated by the “and” logical operator must equate to true in order for Flash to return a value of true for the compound condition. If any one or all of the simple conditions are false, Flash will return a value of false for the compound condition.

```
var value1:Number = 1;
var value2:Number = 2;

if(value1 == value1 && value2 == value2){
    trace("value1 is equal to value1 and value 2 is equal to value2");
}
```

The second logical operator || is referred to as “or”.

In order to get a value of true returned for a compound condition using several simple conditions, within the same parenthesis and separated by the “or” logical operator, at least one of the simple conditions in this scenario must equate to true.

```
var value1:Number = 1;
var value2:Number = 2;

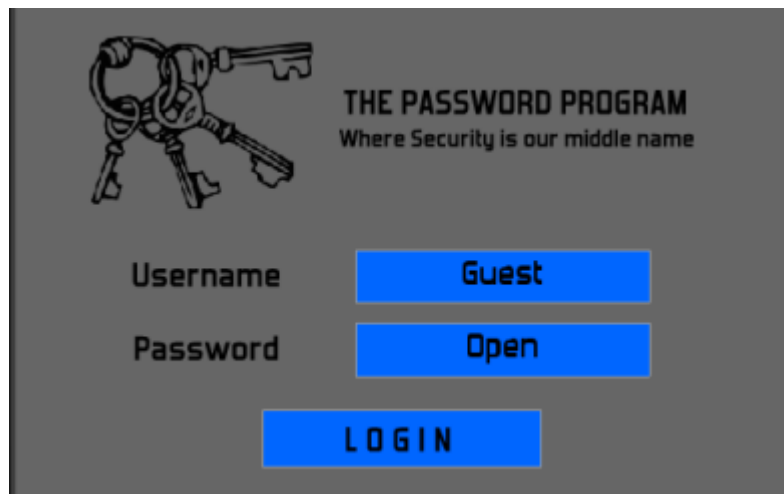
if(value1==value2 || value1 == value1){
    trace("value1 is either equal to value1 or value2");
}
```

Flash will only return a value of false if all simple conditions in the above mentioned scenario equate to false.

```
var value1:Number = 1;
var value2:Number = 2;
var value3:Number = 3;

if(value1==value2 || value2 == value3){
    trace("true");
}else{
    trace("false");
}
```

See passwordProg for an example of compound conditionals.



Appendix

Commonly used Flash commands

trace() Used mainly for development purposes and not included in the final SWF. Expressions included within parenthesis will be evaluated and printed in the Output Window during testing.

setProperty(target, property, value) The target is usually specified with a path eg. `_root.movieClip`. Property is amongst others x scale (`_xscale`), y scale (`_yscale`), x position (`_x`), y position (`_y`). Value can be a specific value or an expression. Accessing an instances properties directly is recommended where possible eg. `movieClip._x = 500`; this elevates the need for using `setProperty()`. See `carMcProperty` and `loaderBar`.

_xmouse Get the x position of the mouse.

_ymouse Get the y position of the mouse.

getBytesLoaded() usually preceded by a path such as `_root.getBytesLoaded()` this will return a value in bytes representing how much of the flash movie has downloaded to a users local harddrive.

getBytesTotal() also usually preceded by a path such as `_root.getBytesTotal()` this will return a value in bytes representing the total size of the flash movie the user is trying to access. `getBytesLoaded()` and `getBytesTotal()` are usually used in conjunction to create a preloader see `loader`.

setMask() A movie clip property used to define a movie clip as being the masked object eg. `bkgMc.setMask('maskMc')` `bkgMc` is the background object that will be masked by the movie clip `maskMc`. Both movieClips `cacheAsBitmap` properties must be checked or set to true either in actionscript or in the movieclips property inspector. For an example of magnification see `magnify`.

loadMovieNum(movie, z-index) used to composite one flash movie on top of another. The movie parameter accepts a string that is the name of the movie to load (including it's path). The z-index parameter is a number representing the place in the stack of composited movies that the current movie will be loaded into. The background movie is always at level 0. see `daffy` for an example of this technique.

attachMovie(existingMovie, newName, z-index) usually preceded with a path eg `_root.attachMovie(_root.movieClip, "newMovieClip", 1)`; The `existingMovie` parameter will accept a movieclips name and a path. The `newName` parameter will accept the new name of the newly created movie clip. This can either be a string or an expression (usually in the case of including this command within a loop). The z-index parameter is a number representing the place in the stack of composited movies that the current movie will be loaded into.

Stage.width A numerical value representing the width of the stage as defined in the Document Properties dialogue. See `scrollingInterface` for an example of this action.

Stage.height A numerical value representing the height of the stage as defined in the Document Properties dialogue. See `scrollingInterface` for an example of this action.

onClipEvent(event){statements} this handler is used to attach actions to a movieclip. The event parameter is can be `enterFrame`, which will execute the statements once for every frame of the movie. Event can also be `load` which will execute the statements only once when the movie is initially loaded. Event can also be `mouseDown` which will execute the statements each time the mouse button (left) is clicked. See `spaceShip` for an example of the `onClipEvent` handler.

int() Converts the numerical value within parenthesis to an integer, by simply removing the decimal value and not by rounding up or down. See `carIntInput`.

startDrag(target, boolean, left, top, right, bottom) initializes a movieclip or button to be draggable. Often used within a clip event handler such as `on(press)`. Target is the name and path of the movie clip to drag. Boolean is either true or false, if true will snap the dragged object's

center to the tip of the mouse. Left, top, right, bottom if specified will constrain the objects draggable area.

stopDrag() Accepts no parameters but stops a movieclip or button currently being dragged from being draggable. Usually used within a clip event handler such as **on(release)**.

Key.isDown(KEY CODE) checks if the key as indicated by it's KEY CODE is depressed.

Key.isUp(KEY CODE) checks if the key as indicated by it's KEY CODE is not depressed.

hitTest() Generally the hitTest() method is used in the following format

myMovieClip.**hitTest**(targetMovieClip) where myMovieClip is the name of the movie clip (accessed most commonly using dot notation) who's hitTest() method you are accessing. targetMovieClip is the name of the movie clip (accessed most commonly using dot notation) that might possibly be colliding with myMovieClip. hitTest() returns a boolean of either true or false, making it easy to use in a conditional expression.

Definitions

1. A program or a script is a set of instructions given to a computer in a certain language eg ActionScript. Programs are usually made up of one or many statements.
2. A statement is a complete instruction issued to a computer via it's programming language. Statements usually consist of two or more parts, a command and an expression.
3. A command is like a verb it tells the Flash player to do something.
- 3) An expression is a way of indicating a value to a program. An expression can also evaluate to something for example the expression 2+5 evaluates to 7.
- 4) Creating code is the process of programming statements.
- 5) Expressions generally evaluate to one of several different data types strings, floating point numbers (floats), booleans or integers.
- 6) Script is another way of saying program
- 7) To execute code is to have the software that reads the code compile the code and hopefully make it do what you intended it to do.
- 8) Algorithms. The logical process by which a problem can be solved or a decision made by a computer program.
- 9) AI (artificial intelligence) Set of Algorithms capable of making complex logical decisions.
- 10) Client. In programming sense refers to a person or user on a station (or the actual station itself) on which an end program is running in a series of other interlinked programs (either remotely or locally).
- 11) Collision Detection. Act of identifying the spacial intersection of two or more objects and/or points (Please note that even though two points cannot collide in reality in a computer program they can).
- 12) Collision Reaction. What happens after a collision has been detected.
- 13) Multi-player server/socket server/multi-user server. Software running on a remote computer. In terms of Flash, it allows data to be received, processed and sent to various clients connected to it.
- 14) Realtime. An indication of how time is dealt with by a program, through the programs ability to process information instantaneously.
- 15) Render. In terms of Flash, the drawing of an object on screen in realtime.
- 16) Source code. The original file/s and/or code containing a programs logic. In Flash the file source and source code is compiled into a new file ie a swf. This swf is what is generally distributed to clients. The source file and source code is either in a .fla file or respectively in a .fla file and a .as file.