# Essential Javascript -- A Javascript Tutorial

By Patrick Hunlock

Javascript is an interpreted language with a C like syntax. While many people brush the language off as nothing more than a browser scripting language, it actually supports many advanced concepts such as object-oriented-programing, recursion, lambda, and closures. It's a very approachable language for the beginner that quickly scales to be as powerful a tool as your skills allow.

This reference will cover the basic language constructs. This is not a beginner's guide to programming. This article focuses on bringing people who already know another programming language up to speed on Javascript methodology. Additionally, this is not an exhaustive language definition, it is a broad overview that will occasionally focus in on some more advanced concepts. It's here to get you started, other articles will focus on making you an expert.

## Getting Started

To dive into Javascript all you need is a simple text-editor and a browser. In windows, you can use notepad under your accessories and Linux and mac users have a similar editor. Simply create a blank HTML page as such…

```
<html>
  <head>
     <title>Learning Javascript</title>
  </head>
  <body>
     <p>Hello World!
  </body>
</html>
```

Save the file then in your browser type in the file name you just created to see the results. Javascript is interpreted so any changes you make to this file will show up instantly in the browser the moment you hit the reload button.

## In-Line Javascript

To define a Javascript block in your web page, simply use the following block of HTML.

```
<script type='text/javascript'>
// Your script goes here.
</script>
```

You can place these script blocks anywhere on the page that you wish, there are some rules and conventions however. If you are generating dynamic content as the page loads you will want the script blocks to appear where you want their output to be. For instance, if I wanted to say "Hello World!" I would want my script block to appear in the <body> area of my web page and not in the <head> section.

Unless your scripts are generating output as the page loads, good practice says that you should place your scripts at the very bottom of your HTML. The reason for this is that each time the browser encounters a <script> tag it has to pause, compile the script, execute the script, then continue on generating the page. This takes time so if you can get away with it, make sure the browser hits your scripts at the end of the page instead of the start.

## External Javascript

External Javascript is where things get interesting. Any time you have a block of code which you will want to use on several different web pages you should place that block in an external Javascript file. The clock on the upper right-hand corner of this page is a good example. The clock appears on almost every page on this site and so it is included in my "common.js" file. Every web-page on the site will load this file and so the clock is available to all of my web-pages.

There's nothing fancy about an external Javascript file. All it is, is a text file where you've put all your Javascript. Basically everything that would ordinarily go ***between*** the <script> tags can go in your external file. Note that between was stressed, you can not have the <script> </script> tags themselves in your external file or you will get errors.

The biggest advantage to having an external Javascript file is that once the file has been loaded, the script will hang around the browser's cache which means if the Javascript is loaded on one page then it's almost a sure thing that the next page on the site the user visits will be able to load the file from the browser's cache instead of having to reload it over the Internet (This is an incredibly fast and speedy process).

Including an external file is basically the same as doing an in-line script, the only difference is that you specify a filename, and there's no actual code between <script> and </script>...

```
<script type='text/javascript' src='common.js'></script>
```

When the browser encounters this block it will load common.js, evaluate it, and execute it. Like in-line scripts above you can place this block anywhere you need the script to be and like in-line scripts you should place these as close to the bottom of the web-page as you can get away with.

The only difference between in-line Javascript blocks and external Javascript blocks is that an external Javascript block will pause to load the external file. If you discount that one thing, there's no procedural difference between the two!

## Javascript is case sensitive.

It should also be noted, before we begin, that Javascript is extremely case sensitive so if you're trying to code along with any examples make sure lowercase is lowercase and uppercase is uppercase. For the most part Javascript is also a camel-cased language. That is, if you're trying to express more than one word you will eliminate the spaces, leave the first letter uncapitalized and capitalize the first letter of each word. Thus "get element by id" becomes "getElementByID".

By contrast, HTML itself is NOT case sensitive.

# Output (writeln)

One of the most important things to do when learning a new language is to master basic input and output which is why hello world has become almost a cliché in programming textbooks. For Javascript you need three hello worlds because there are three ways to communicate with the user, each increasingly more useful than the last.

The first method is to use the `document.writeln(string)` command. This can be used while the page is being constructed. After the page has finished loading a new `document.writeln(string)` command will delete the page in most browsers, so use this only while the page is loading. Here's how a simple web-page will look...

```
<html>
  <head>
  </head>
  <body>
     <script type='text/javascript'>
        document.writeln('Hello World!');
     </script>
  </body>
</html>
```

As the page is loading, Javascript will encounter this script and it will output "Hello World!" exactly where the script block appears on the page.

The problem with *writeln* is that if you use this method after the page has loaded the browser will destroy the page and start constructing a new one.

For the most part, document.writeln is useful only when teaching yourself the language. Dynamic content during page load is better served by the server-side scripting languages. That said, *document.writeln* is very useful in pre-processing forms before they're sent to the server -- you can basically create a new web-page on the fly without the need to contact the server.

# Output (alert)

The second method is to use a browser alert box. While these are incredibly useful for debugging (and learning the language), they are a horrible way to communicate with the user. Alert boxes will stop your scripts from running until the user clicks the OK button, and it has all the charm and grace of all those pop-up windows everyone spent so many years trying to get rid of!

```
<html>
  <head>
  </head>
  <body>
     <script type='text/javascript'>
        alert('Hello World!');
     </script>
  </body>
</html>
```

# Output (getElementById)

The last method is the most powerful and the most complex (but don't worry, it's really easy!).

Everything on a web page resides in a box. A paragraph (<P>) is a box. When you mark something as bold you create a little box around that text that will contain bold text. You can give each and every box in HTML a unique identifier (an ID), and Javascript can find boxes you have labeled and let you manipulate them. Well enough verbiage, check out the code!

```
<html>
  <head>
  </head>
  <body>
     <div id='feedback'></div>
     <script type='text/javascript'>
        document.getElementById('feedback').innerHTML='Hello World!';
     </script>
  </body>
</html>
```

The page is a little bigger now but it's a lot more powerful and scalable than the other two. Here we defined a division  and named it "*feedback*". That HTML has a name now, it is unique and that means we can use Javascript to find that block, and modify it. We do exactly this in the script below the division! The left part of the statement says on this web page (*document*) find a block we've named "*feedback*" ( *getElementById('feedback')* ), and change its HTML (*innerHTML*) to be *'Hello World!'*.

We can change the contents of '*feedback*' at any time, even after the page has finished loading (which document.writeln can't do), and without annoying the user with a bunch of pop-up alert boxes (which alert can't do!).

It should be mentioned that innerHTML is not a published standard. The standards provide ways to do exactly what we did in our example above. That mentioned, innerHTML is supported by every major Browser and in addition innerHTML works faster, and is easier to use and maintain. It's, therefore, not surprising that the vast majority of web pages use innerHTML over the official standards.

While we used "Hello World!" as our first example, its important to note that, with the exception of <script> and <style>, you can use full-blown HTML. Which means instead of just Hello World we could do something like this…

```
<html>
  <head>
  </head>
  <body>
     <div id='feedback'></div>
     <script type='text/javascript'>
        document.getElementById('feedback').innerHTML='<P><font color=red>Hello
World!</font>';
     </script>
  </body>
</html>
```

In this example, *innerHTML* will process your string and basically redraw the web page with the new content. This is a VERY powerful and easy to use concept. It means you can basically take an empty HTML element (which our feedback division is) and suddenly expand it out with as much HTML content as you'd like.

## Input (One Click To Rule Them All)

Input, of course, is a little more complicated. For now we'll just reduce it to a bare click of the mouse.

If everything in HTML is a box and every box can be given a name, then every box can be given an event as well and one of those events we can look for is "*onClick*". Lets revisit our last example...

```
<html>
  <head>
  </head>
  <body>
     <div id='feedback' onClick='goodbye()'>Users without Javascript see
this.</div>
     <script type='text/javascript'>
        document.getElementById('feedback').innerHTML='Hello World!';
        function goodbye() {
           document.getElementById('feedback').innerHTML='Goodbye World!';
        }
     </script>
  </body>
</html>
```

Here we did two things to the example, first we added an "*onClick*" event to our feedback division which tells it to execute a function called *goodbye()* when the user clicks on the division. A function is nothing more than a named block of code. In this example goodbye does the exact same thing as our first hello world example, it's just named and inserts 'Goodbye World!' instead of 'Hello World!'.

Another new concept in this example is that we provided some text for people without Javascript to see. As the page loads it will place "Users without Javascript will see this." in the division. If the browser has Javascript, and it's enabled then that text will be immediately overwritten by the first line in the script which looks up the division and inserts "Hello World!", overwriting our initial message. This happens so fast that the process is invisible to the user, they see only the result, not the process. The *goodbye()* function is not executed until it's explicitly called and that only happens when the user clicks on the division.

While Javascript is nearly universal there are people who surf with it deliberately turned off and the search bots (googlebot, yahoo's slurp, etc) also don't process your Javascript, so you may want to make allowances for what people and machines are-not seeing.

# Input (User Input)

Clicks are powerful and easy and you can add an onClick event to pretty much any HTML element, but sometimes you need to be able to ask for input from the user and process it. For that you'll need a basic form element and a button…

```
<input id='userInput' size=60> <button onClick='userSubmit()'>Submit</button><BR>
<P><div id='result'></div>
```

Here we create an input field and give it a name of *userInput*. Then we create a HTML button with an *onClick* event that will call the function *userSubmit()*. These are all standard HTML form elements but they're not bound by a <form> tag since we're not going to be submitting this information to a server. Instead, when the user clicks the submit button, the *onClick* event will call the *userSubmit()* function…

```
<script type='text/javascript'>
function userSubmit() {
   var UI=document.getElementById('userInput').value;
   document.getElementById('result').innerHTML='You typed: '+UI;
}
</script>
```

Here we create a variable called *UI* which looks up the input field *userInput*. This lookup is exactly the same as when we looked up our feedback division in the previous example. Since the input field has data, we ask for its *value* and place that value in our *UI* variable. The next line looks up the *result* division and puts our output there. In this case the output will be "You Typed: " followed by whatever the user had typed into the input field.

We don't actually need to have a submit button. If you'd like to process the user input as the user types then simply attach an *onKeyup* event to the input field as such…

```
<input id='userInput' onKeyUp="userSubmit()" size=60><BR>
<P><div id='result'></div>
```

There's no need to modify the *userSubmit()* function. Now whenever a user presses a key while the *userInput* box has the focus, for each keypress, *userSubmit()* will be called, the value of the input box retrieved, and the *result* division updated.

# Javascript is an Event Driven Language

As you can tell from the input examples, Javascript is an event driven language which means your scripts react to events you set up. Your code isn't running all the time, it simply waits until an event starts something up! Going into all the Javascript events is beyond the scope of this document but here's a short-list of common events to get you started.

| Event | Description |
|---|---|
| onAbort | An image failed to load. |
| onBeforeUnload | The user is navigating away from a page. |
| onBlur | A form field lost the focus (User moved to another field) |
| onChange | The contents of a field has changed. |
| onClick | User clicked on this item. |
| onDblClick | User double-clicked on this item. |
| onError | An error occurred while loading an image. |
| onFocus | User just moved into this form element. |
| onKeyDown | A key was pressed. |
| onKeyPress | A key was pressed OR released. |
| onKeyUp | A key was released. |
| onLoad | This object (iframe, image, script) finished loading. |
| onMouseDown | A mouse button was pressed. |
| onMouseMove | The mouse moved. |
| onMouseOut | A mouse moved off of this element. |
| onMouseOver | The mouse moved over this element. |
| onMouseUp | The mouse button was released. |
| onReset | A form reset button was pressed. |
| onResize | The window or frame was resized. |
| onSelect | Text has been selected. |
| onSubmit | A form's Submit button has been pressed. |
| onUnload | The user is navigating away from a page. |

These events can be attached to most any HTML tag or form element. Of them all *onClick* will probably be what you end up using most often.

# Comments

Javascript supports two types of comments. Double-slashes (//) tell javascript to ignore everything to the end of the line. You will see them used most often to describe what is happening on a particular line.

```
var x=5; // Everything from the // to end of line is ignored(*)
var thingamajig=123.45; // 2 times the price of a whatsit.
```

Block quotes begin a comment block with a slash-asterisk (/*) and Javascript will ignore everything from the start of the comment block until it encounters an asterisk-slash (*/). Block quotes are useful for temporally disabling large areas of code, or describing the purpose of a function, or detailing the purpose and providing credits for the script itself.

```
function whirlymajig(jabberwocky) {
   /* Here we take the jabberwocky and insert it in the gire-gimble,
      taking great care to observe the ipsum lorum!   For bor-rath-outgrabe!
      We really should patent this! */

   return (jabberwocky*2);
}
```

You should note that while comments are useful for maintaining the code, they are a liability itself in Javascript since they will be transmitted along with the code to each and every page load, which can create substantial bandwidth penalties and increase the load time of your page for users.

This doesn't mean you shouldn't comment your code, just that once your code is "finished" you should make a backup copy with the comments, then strip out all the comments in the file which is actually sent to the user. You can automate this process with a minimizing application which you can find at http://www.crockford.com/javascript/jsmin.html and an on-line javascript version at http://fmarcia.info/jsmin/test.html .

The result of minimizing your Javascript is a tiny, compact file which is a fraction of the size of the original which will save you bandwidth and provide speedier page-load time for your visitors. However the result is also a very unmaintainable source-code mess which is why you should keep a separate, unminimized (and heavily commented) version of the original file.

(*) Of special consideration, you should note that the browser itself is ALWAYS looking for a </script> tag to mark the end of your Javascript and if it finds that tag, intact, in-one-piece, be it in a string or a comment, it is going to stop processing Javascript at that point and restart-processing HTML.

```
var x=5;

/* The browser will break the Javascript when it sees this </script> tag.
   Everything from tag forward is now being processed as HTML!
   This is a bad thing!  To avoid this you need to avoid using this
   tag anywhere in your Javascript, and if
   you must have it, you should break the string out like this... */

  document.writeln('</scr'+'ipt>');
```

# Variables

Javascript is not a strongly typed language which means you rarely have to concern yourself with the type of data a variable is storing, only what the variable is storing and in Javascript, variables can store anything, even functions.

```
var thisIsAString = 'This is a string';
var alsoAString = '25';
var isANumber = 25;
var isEqual = (alsoAString==isANumber);  // This is true, they are both 25.
var isEqual = (alsoAString===isANumber); // False one is a number, the other a string.
var concat=alsoAString + isANumber;      // concat is now 2525
var addition=isANumber + isANumber;      // addition is now 50
var alsoANumber=3.05;     // is equal to 3.05 (usually).
var floatError=0.06+0.01; // is equal to 0.06999999999999999
var anExponent=1.23e+3;   // is equal to 1230
var hexadecimal = 0xff;   // is equal to 255.
var octal = 0377;         // is equal to 255.
var isTrue = true;   // This is a boolean, it can be true or false.
var isFalse= false;  // This is a boolean, it can be true or false
var isArray = [0, 'one', 2, 3, '4', 5]; // This is an array.
var four = isArray[4]; // assign a single array element to a variable.
                       // in this case four = '4'
var isObject = { 'color': 'blue',  // This is a Javascript object
                 'dog': 'bark',
                 'array': [0,1,2,3,4,5],
                 'myfunc': function () { alert('do something!'); }
               }
var dog = isObject.dog;  // dog now stores the string 'bark';
isObject.myfunc(); // creates an alert box with the value "do something!"
var someFunction = function() {
                     return "I am a function!";
                   }
var alsoAFunction = someFunction; //No () so alsoAFunction becomes a function
var result = alsoAFunction(); // alsoAFunction is executed here because ()
                              // executes the function so result stores the
                              // return value of the function which is
                              // "I am a function!"
```

A variable may not be a Javascript reserved word or begin with a number or any symbol other than $ or _. In Internet explorer you should also avoid variable names with the same name as html elements you have named. For instance…

```
var someDiv = document.getElementByID('someDiv');
```

…will cause problems in Internet Explorer because the variable name and the division name are identical.

In recent years a convention has formed around the use of the $ symbol as various libraries like Prototype and JQuery use it to look up a named HTML element. For most purposes if you see $('something') in Javascript you should read that as being *document.getElementById('something')*. This is not standard Javascript, in order for *$('something')* to work, you need to be using a Javascript framework which will define $ as doing something (Like JQuery, Prototype, etc).

The use of a leading underscore (_) is generally useful to indicate a global variable or a variable that has been set outside the current scope.

A final consideration on variables is that functions themselves can be defined like, and act like variables. Once a function has been defined it can be passed to other functions as an argument (A process knows as lambda), or assigned to other variables just like a string, array or any other Javascript object. Generally if you use a function without trailing parenthesis (), the function is treated like a variable and can be passed and assigned. Trailing parenthesis INVOKE the function, executing it and passing back the return value (if any).

Please note that this is a very broad summary overview of Javascript's data types. For more information please see the other articles in this series (listed at the top of the page) which go into exhaustive detail on each Javascript type.

# Variable Scope

Variables in Javascript have FUNCTION scope. That is, all variables are global unless they are explicitly defined inside a function and even then child-functions have access to their parent's variables. If a function defines a new variable WITHOUT using the *var* keyword, that variable will be **global** in scope.

```
var global = 'this is global';
function scopeFunction() {
   alsoGlobal = 'This is also global!';
   var notGlobal = 'This is private to scopeFunction!';

   function subFunction() {
      alert(notGlobal); // We can still access notGlobal in this child function.

      stillGlobal = 'No var keyword so this is global!';
      var isPrivate = 'This is private to subFunction!';
   }

   alert(stillGlobal); // This is an error since we haven't executed subfunction
   subFunction();      // execute subfunction
   alert(stillGlobal); // This will output 'No var keyword so this is global!'
   alert(isPrivate);   // This generate an error since isPrivate is private to
                       // subfunction().

   alert(global);      // outputs: 'this is global'
}
alert(global);      // outputs: 'this is global'
alert(alsoGlobal);  // generates an error since we haven't run scopeFunction yet.
scopeFunction();
alert(alsoGlobal);  // outputs: 'This is also global!';
alert(notGlobal);   // generates an error.
```

The concept that a variable will continue to exist, and can be referenced after the function that created it has ceased executing is known as CLOSURE. In the above example, *stillGlobal*, and *alsoGlobal* can be considered closures because they persist after the function that creates them has ceased to operate. You can do some pretty fancy stuff with it later on, but it's not terribly hard to understand once you associate it with creating a global scoped variable inside a function.

# Special Keywords

Javascript has a few pre-defined variables with special meaning.

*NaN* -- Not a Number (Generated when an arithmetic operation returns an invalid result). *NaN* is a weird construct. For one, it is NEVER equal to itself so you can't simply check to see if *3/'dog' == 'NaN'*. You must use the construct *isNaN(3/dog)* to determine if the operation failed. In boolean operations *NaN* evaluates to false, however 0 also evaluates to false so use *isNaN*. Since *NaN* is never equal to itself you can use this simple trick as well:

if (result != result) { alert('Not a Number!'); }

*Infinity* is a keyword which is returned when an arithmetic operation overflows Javascript's precision which is in the order of 300 digits. You can find the exact minimum and maximum range for your Javascript implementation using *Number.MAX_VALUE* and *Number.MIN_VALUE*.

*null* is a reserved word that means "empty". When used in boolean operation it evaluates to false.

Javascript supports *true* and *false* as boolean values.

If a variable hasn't been declared or assigned yet (an argument to a function which never received a value, an object property that hasn't been assigned a value) then that variable will be given a special *undefined* value. In boolean operations *undefined* evaluates as *false*. Here's an example…

```
function doAlert(sayThis) {
   if (sayThis===undefined) {   // Check to see if sayThis was passed.
      sayThis='default value';  // It wasn't so give it a default value
   }                            // End check
   alert(sayThis);              // Toss up the alert.
}
```

# Arithmetic Operators

There's nothing particularly exotic about the way Javascript does arithmetic.

| Operator | Description |
|---|---|
| + | Addition (also string concatention) |
| - | Subtration (also unary minus) |
| * | Multiplication |
| / | Division |
| % | Remainder of division (or modulus) |
| ++ | pre or post increment |
| -- | pre or post decrement |

++ and -- are C style operators their position around the variable they are operating on is important. If the operator appears BEFORE the variable they will be evaluated immediately, if they appear AFTER the variable they will be evaluated only after the entire line has been resolved. For instance…

```
var x = 5;
 var y = x++;      // y=5, x=6
 var x = 5;
 var y = ++x;      // y=6, x=6
```

In the first example y is assigned the value of x (5) and then x is incremented by one because the ++ operator appears AFTER x. In the second example, x is incremented by one first because ++ appears BEFORE the x, so y is given the value 6.

How this works between C and PhP and Javascript can be somewhat different. There was a most excellent discussion about this on reddit if you'd like to delve into those differences.

Additional shorthand operators exist when working on the same variable. For instance…

```
x = x + 5;  // is the same as...
x += 5;
```

# Logical and Comparison Operators

Javascript supports equality checking (==) and *identity* checking (===). Equality checks for equality regardless of type. Therefore 25 and '25' will evaluate as true. Identity checking checks not only for equality but type equality as well so 25 and '25' will evaluate as false because, while both are 25, one is a string and the other a number. Note that a single equal sign is an assignment statement! x=5 will assign 5 to x, while x==5 will see if x is equal to 5, and x===5 will check to see if x is identical to 5.

In addition to == and ===, you can check for not equal (!=) and not identical (!==).

| Operator | Description |
|---|---|
| = | Assignment x=5; // assigns 5 to x |
| == | Equality, is x==5? |
| === | Identity, is x 5 and a number as well? |
| != | Not equal, is x unequal to 5? |
| !== | Not identical, is x unequal to the Number 5? |
| ! | Not, if not(false) is true. |
| \|\| | OR, is (x==5) OR (y==5) |
| && | And, is (x==5) AND (y==5) |
| < | Less than. is x less than 5? |
| <= | Less than or equal. is x less than or equal to 5? |
| > | Greater than. is x greater than 5? |
| >= | Greater than or qual. is x greater than or equal to 5? |

# Conditionals: IF

The *if* statement lets you execute a block of code if some test is passed.

```
var x=5;
if (x==5) {
   alert('x is equal to 5!');
}
```

You can also use an *else* clause to execute code if the test fails.

```
var x=5;
if (x==5) {
   alert('x is equal to 5!');
} else {
   alert('x is not equal to 5!');
}
```

An *elseif* statement also exists which allows for better formating of long conditional tests.

```
var x=5;
if (x==1) {
   alert('x is equal to 1!');
} else if (x==2) {
   alert('x is equal to 2!');
} else if (x==5) {
   alert('x is equal to 5!');
} else {
   alert('x isn't 1, 2 or 5!');
}
```

# Conditionals: SWITCH

If you're going to be doing a large number of tests, it makes sense to use a switch statement instead of nested ifs. Switches in javascript are quite powerful, allowing evaluations on both the switch and the case.

```
var x=5;
switch (x) {
   case 1: alert('x is equal to 1!'; break;
   case 2: alert('x is equal to 2!'; break;
   case 5: alert('x is equal to 5!'; break;
   default: alert("x isn't 1, 2 or 5!");
}
```

Note that if you omit the *break* statement that ALL of the code to the end of the switch statement will be executed. So if x is actually equal to 5 and there is no break statement, an alert for "x is equal to 5" will appear as well as an alert for "x isn't 1,2, or 5!".

Sometimes it makes more sense to do the evaluation in the case statement itself. In this case you'd use true, false, or an expression which evaluates to true or false in the switch statement.

```
var x=5;
switch (true) {
   case (x==1): alert('x is equal to 1!'; break;
   case (x==2): alert('x is equal to 2!'; break;
   case (x==5): alert('x is equal to 5!'; break;
   default: alert("x isn't 1, 2 or 5!");
}
```

## Conditionals: Shorthand Assignment

Javascript supports more advanced constructs. Often you will see code like the following…

```
function doAddition(firstVar, secondVar) {
   var first = firstVar  ||  5;
   var second= secondVar || 10;
   return first+second;
}
doAddition(12);
```

Here Javascript uses a logical OR (||) to determine if the passed variables actually have a value. In the example we call *doAddition* with a value of 12 but we neglect to pass a second argument. When we create the *first* variable *firstVar* is a non-falsey value (IE it's actually defined) so Javascript assigns firstVar to first. *secondVar* was never passed a value so it is *undefined* which evaluates to *false* so here the variable *second* will be assigned a default value of 10.

You should note that zero evaluates as false so if you pass zero as either *firstVar* or *secondVar* the default values will be assigned and NOT zero. In our example above it is impossible for *first* or *second* to be assigned a zero.

In psuedo code...

```
var someVariable = (assign if this is truthy) || (assign this if first test evaluates false)
```

## Conditionals: Ternary Operators

Ternary operators are a shorthand if/else block who's syntax can be a bit confusing when you're dealing with OPC (Other People's Code). The syntax boils down to this.

```
var userName = 'Bob';
var hello = (userName=='Bob') ? 'Hello Bob!' : 'Hello Not Bob!';
```

In this example the statement to be evaluated is *(userName=='Bob')*. The question marks ends the statement and begins the conditionals. If *UserName* is, indeed, Bob then the first block *'Hello Bob!'* will be returned and assigned to our *hello* variable. If *userName* isn't Bob then the second block ('Hello Not Bob!') is returned and assigned to our *hello* variable.

In psudeo code…

```
var someVariable = (condition to test) ? (condition true) : (condition false);
```

The question mark (?) and colon (:) tend to get lost in complex expressions as you can see in this example taken from wikipedia (but which will also work in Javascript if the various variables are assigned...)

```
for (i = 0; i < MAX_PATTERNS; i++)
   c_patterns[i].ShowWindow(m_data.fOn[i] ? SW_SHOW : SW_HIDE);
```

So while quick and efficient, they do tend to reduce the maintainability/readability of the code.

## Loops: FOR

The *for* loop follows basic C syntax, consisting of an initialization, an evaluation, and an increment.

```
for (var i=0; (i<5); i++) {
   document.writeln('I is equal to '+i+'<br>');
}
// outputs:
// I is equal to 0
// I is equal to 1
// I is equal to 2
// I is equal to 3
// I is equal to 4
```

This is actually an extreme simplification of what a for statement can do. On the other end of the spectrum, consider this shuffle prototype which will randomly shuffle the contents of an array. Here, everything is defined, and executed within the context of the for statement itself, needing no additional block to handle the code.

```
Array.prototype.shuffle = function (){
   for(var rnd, tmp, i=this.length; i; rnd=parseInt(Math.random()*i), tmp=this[--
i], this[i]=this[rnd], this[rnd]=tmp);
};
```

# Loops: FOR/IN

Javascript has a variant of the for loop when dealing with Javascript objects.

Consider the following object…

```
var myObject = { 'animal'  : 'dog',
                 'growls'  : true,
                 'hasFleas': true,
                 'loyal'   : true }
```

We can loop through these values with the following construct.

```
var myObject = { 'animal'  : 'dog',
                 'growls'  : true,
                 'hasFleas': true,
                 'loyal'   : true }

for (var property in myObject) {
   document.writeln(property + ' contains ' + myObject[property]+'<br>');
}
// Outputs:
// animal contains dog
// growls contains true
// hasFleas contains true
// loyal contains true
```

What this essentially does is assign the property name to the variable *property*. We can then access *myObject* through an associative array style syntax. For instance the first itteration of the loop assigns *animal* to *property* and *myObject["animal"]* will return dog.

There is a big caveat here in that properties and methods added by prototyping will also show up in these types of loops. Therefore it's best to always check to make sure you are dealing with data and not a function as such…

```
for (var property in myObject) {
   if (typeof(myObject[property]) != 'function') {
      document.writeln(property + ' contains ' + myObject[property]+'<br>');
   }
}
```

The typeof check to screen out functions will ensure that your for/in loops will extract only data and not methods that may be added by popular javascript libraries like Prototype.

# Loops: WHILE

*while* loops in Javascript also follow basic C syntax and are easy to understand and use.

The *while* loop will continue to execute until its test condition evaluates to false or the loop encounters a *break* statement.

```
var x = 1;
while (x<5) {
   x = x +1;
}
var x = 1;
while (true) {
   x = x + 1;
   if (x>=5) {
      break;
   }
}
```

Sometimes it makes more sense to evaluate the test condition at the end of the loop instead of the beginning. So for this Javascript supports a *do/while* structure.

```
var x=1;
do {
   x = x + 1;
} while (x < 5);
```

# Bringing It All Together

So now you know how to do basic input and output, how to do conditional branching and how to do loops. That's pretty much everything you need to know about any programming language to get started! So lets wrap everything we've done so far into one simple web application.

First we'll define our web page.

```
<html>
   <head>
      <title>My First Javascript</title>
   </head>
   <body>
      Hello World!
   </body>
</html>
```

Now to this we will add an input line, a drop down box, and a submit button. Notice we give an ID to the form elements, and have the submit button call a function when it's clicked. A function which we will write later. We've also added some descriptive text.

```
<html>
   <head>
      <title>My First Javascript</title>
   </head>
   <body>
      Hello World!
      <p>Say what? <input id="sayThis" size=40>
      <P>How many times? <select id='howMany'>
                            <option value=1>1</option>
                            <option value=5 selected>5</option>
                            <option value=10>10</option>
                            <option value=20>20</option>
                         </select>
      <p><button onClick='doLoop()'>Do It!</button>
   </body>
</html>
```

Now we need to add a division where the output will occur. We'll add this right below the form elements, specifically the "Do It!" button.

```
<html>
   <head>
      <title>My First Javascript</title>
   </head>
   <body>
      Hello World!
      <p>Say what? <input id="sayThis" size=40>
      <P>How many times? <select id='howMany'>
                            <option value=1>1</option>
                            <option value=5 selected>5</option>
                            <option value=10>10</option>
                            <option value=20>20</option>
                         </select>
      <p><button onClick='doLoop()'>Do It!</button>
      <p><div id="results"></div>
   </body>
</html>
```

The script for this will be simple enough. When the button is clicked the *doLoop()* function will be called. doLoop() will lookup the value of the *sayThis* input field, then repeat that *howMany* times.

Here's the script itself, separate from the HTML...

```
  function doLoop() {
      var sayWhat = document.getElementById('sayThis').value;
      var maxLoop = document.getElementById('howMany').value;
      var str = ''; // where we'll store our output temporarily.
      for (var i=1; (i<=maxLoop); i++) {
         str=str+i+':'+sayWhat+'<br>';
      }
      document.getElementById("results").innerHTML=str;
   }
```

And the entire application, all together.

```
<html>
   <head>
      <title>My First Javascript</title>
   </head>
   <body>
      Hello World!
      <p>Say what? <input id="sayThis" size=40>
      <P>How many times? <select id='howMany'>
                              <option value=1>1</option>
                              <option value=5 selected>5</option>
                              <option value=10>10</option>
                              <option value=20>20</option>
                           </select>
      <p><button onClick='doLoop()'>Do It!</button>
      <p><div id="results"></div>
      <script type='text/html'>
         function doLoop() {
            var sayWhat = document.getElementById('sayThis').value;
            var maxLoop = document.getElementById('howMany').value;
            var str = ''; // where we'll store our output temporarily.
            for (var i=1; (i<=maxLoop); i++) {
               str=str+i+':'+sayWhat+'<br>';
            }
            document.getElementById("results").innerHTML=str;
         }
      </script>
   </body>
</html>
```

# DHTML: Dynamic HTML

The above example is what's technically considered Dynamic HTML (or DHTML) because the contents of the page dynamically change based on user interaction after the page has finished loading. Going into great detail on DHTML is beyond the scope of this article but there is one small, trivial jump that can really change how you see HTML and Javascript.

This page has pulled an HTML element with the use of document.getElementById. Once we've had this assigned to a variable we've been able to manipulate the contents of that element with innerHTML, and in the case of forms find the value of the element with the value property.

There is also a style property which lets you access, and change, the CSS styles for that element. There is a nice CSS reference at IloveJackDaniels ( http://www.ilovejackdaniels.com/cheat-sheets/css-cheat-sheet/ ). Most all of the properties you can access are listed on the left and right margin of the cheat sheet (the interior/brown areas aren't applicable). The only real trick is that where you see a dash in the CSS name, for Javascript you need to remove the dash and capitalize the next letter. So the CSS style *background-color* becomes *backgroundColor* in Javascript.

Here's an example which will change the background color of a division when you click on it's contents.

```
<div id="example" onClick="colorize()">Click on this text to change the
background color</div>
<script type='text/javascript'>
   function colorize() {
      var element = document.getElementById("example");
      element.style.backgroundColor='#800';
   }
</script>
```

Here we create a division with a name of *example* which will call a function called *colorize()* when that division is clicked. *colorize()* will lookup the *example* division and assign it to the Javascript variable *element*. Next, we assign a color of #800 (burgundy) to the element's *style.backgroundColor* property.

This actually made the text hard to read so in the next example we'll add another line to change the color of the text to white.

```
<div id="example" onClick="colorize()">Click on this text to change the
background color</div>
<script type='text/javascript'>
   function colorize() {
      var element = document.getElementById("example");
      element.style.backgroundColor='#800';
      element.style.color='white';
   }
</script>
```

And lets go ahead and center the text as well.

```
<div id="example" onClick="colorize()">Click on this text to change the
background color</div>
<script type='text/javascript'>
   function colorize() {
      var element = document.getElementById("example");
      element.style.backgroundColor='#800';
      element.style.color='white';
      element.style.textAlign='center';
   }
</script>
```

The manipulation of an HTML element's CSS styles is incredibly powerful! You can hide, or display elements with the style.display property (*style.display=none*=invisible, *style.display=block*=visible) You can let the elements float over the page with *style.position* (absolute/relative/fixed), you can change their position on the page by setting *style.left* and *style.top*. The possibilities are literally endless.

# Conclusion

As stated at the beginning of this article, this document is intended to bring a programmer up to speed in Javascript, and is not an exhaustive review of the language. This article is a part of a larger series of reference articles (listed at the top of the page). As you master the fundamentals introduced on this page, it's recommended that you visit the other reference articles to master the details of the language.

Javascript is a surprisingly deep and well considered language (with a few notable and notorious exceptions). It is simple enough for beginners to programming to understand and scales to become as advanced as you need it to be. While it may be frustrating to work around the various idiosyncrasies of all the browser models and versions, Javascript itself is a joy to work in, and when developing web-applications, the least and most unobtrusive of all the problems you will have to surmount.

# Additional Resources

While there are many good Javascript books available. The general consensus is that the best introduction and reference to Javascript is Javascript: The Definitive Guide by David Flanagan. This book is endorsed by the Usenet group comp.lang.javascript as well as by Douglas Crockford and other notable people of influence within the Javascript community.

# Version

This document is current as of Firefox 2.03, IE 7.0, JSCRIPT 5.6, ECMA-262 Edition 3, Javascript 1.7

# License