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 <div> 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.