A (Re)-Introduction to JavaScript

Simon Willison - http://simonwillison.net/

Notes made for a 3 hour tutorial at ETech in San Diego, March 6th 2006

INTRODUCTION

My name's Simon Willison, and the title of this session is "A re-introduction to JavaScript".

Why a re-introduction? Because JavaScript has a reasonable claim to being the world's most misunderstood programming language. While often derided as a toy, beneath its deceptive simplicity lie some powerful language features. The last year has seen the launch of a number of high profile JavaScript applications, showing that deeper knowledge of this technology is an important skill for any web developer.

It's useful to start with an idea of the language's history. JavaScript was created in 1995 by Brendan Eich, an engineer at Netscape, and first released with Netscape 2 early in 1996. It's was originally going to be called LiveScript, but was renamed in an ill-fated marketing decision to capitalise on the popularity of Sun Microsystem's Java language - despite the two having very little in common. This has been a source of confusion ever since.

Microsoft released a mostly-compatible version of the language called JScript with IE 3 several months later. Netscape submitted the language to Ecma International, a European standards organisation, which resulted in the first edition of the Ecmascript standard in 1997. The standard received a significant update as Ecmascript edition 3 in 1999, and has stayed pretty much stable ever since - although edition 4 is currently in the works.

This stability is great news for developers as it's given the various implementations plenty of time to catch up. I'm going to focus almost exclusively on the edition 3 dialect. For familiarity, I will stick with the term JavaScript throughout.

Unlike most programming languages, the JavaScript language has no concept of input or output. It is designed to run as a scripting language in a host environment, and it is up to the host environment to provide mechanisms for communicating with the outside world. The most common host environment is the browser, but JavaScript interpreters can also be found in Adobe Acrobat, Photoshop, Yahoo!'s Widget engine and more. To add some variety, I'll be demonstrating some alternative JavaScript hosts later in the tutorial.

OVERVIEW

Let's start off by looking at the building block of any language: the types. JavaScript programs manipulate values, and those values all belong to a type. JavaScript's types are:

- Numbers
- Strings
- Booleans
- Functions
- Objects

... oh, and Undefined and Null, which are slightly odd. And Arrays, which are a special kind of objects. And Dates and Regular Expressions, which are objects that you get for free. And to be technically accurate, functions are just a special type of object. So the type diagram looks more like this:

- Number
- String
- Boolean
- Object
  - Function
  - Array
  - Date
  - RegExp
- Null
- Undefined

And there are some built in Error types as well. Things are a lot easier if we stick with the first diagram though.

NUMBERS

Numbers in JavaScript are "double-precision 64-bit format IEEE 754 values", according to the spec. This has some interesting consequences. There's no such thing

as an integer in JavaScript, so you have to be a little careful with your arithmetic if you're used to maths in C or Java. Watch out for stuff like:

```
0.1 + 0.2 = 0.30000000000000004
```

The standard numeric operators are supported, including addition, subtraction, modulus (or remainder) arithmetic and so forth. There's also a built in object that I forgot to mention earlier called Math to handle more advanced mathematical functions and constants:

```
Math.sin(3.5);
d = Math.PI * r * r;
```

You can convert a string in to a number using the built-in parseInt() function. This takes the base for the conversion as an optional second argument, which you should always provide:

```
> parseInt("123")
123
> parseInt("010")
8
```

That happened because the parseInt function decided to treat the string as octal due to the leading 0. If you always provide the base you can avoid this problem entirely:

```
> parseInt("123", 10)
123
> parseInt("010", 10)
10
```

If you want to convert a binary number to an integer, just change the base:

```
> parseInt("11", 2)
3
```

JavaScript also has a special value called Not-a-Number. This turns up when you do things like trying to convert a non-numeric string in to a number.

> parseInt("hello", 10)
NaN

NaN is toxic - if you provide it as an input to any mathematical operation the result will also be NaN:

```
> NaN + 5
NaN
```

You can detect is using the built-in isNaN function:

```
> isNaN(NaN)
true
```

JavaScript also has special values for Infinity and negative Infinity:

```
> 1 / 0
Infinity

> -1 / 0
-Infinity
```

STRINGS

Strings in JavaScript are sequences of characters. More accurately, they're sequences of unicode characters, with each character represented by a 16 bit number. This should be welcome news to anyone who has had to deal with internationalisation.

If you want to represent a single character, you just use a string of length 1.

To find the length of a string, access its length property:

```
> "hello".length
5
```

There's our first brush with JavaScript objects! Did I mention that strings are objects too? They have methods as well:

```
> "hello".charAt(0)
h
> "hello, world".replace("hello", "goodbye")
goodbye, world
> "hello".toUpperCase()
HELLO
```

OTHER TYPES

Null means a deliberate no value. Undefined means that a value hasn't even been assigned yet. We'll talk about variables later, but in JavaScript it is possible to declare a variably without assigning a value to it. If you do this, the variable's value is the "undefined" value.

JavaScript has a boolean type, which is either true or false (both of which are keywords). Everything else in the language is either "truthy" or "falsy", which determines how it is treated when used in a boolean context. The rules for truthiness are as follows:

```
0, "", NaN, null, and undefined are falsy. Everything else is truthy.
```

Boolean operations such as &&, || and ! are supported. You can convert any value to it's boolean equivalent by applying not twice:

```
> !!""
false
> !!234
```

```
true
```

VARIABLES

New variables in JavaScript are declared using the var keyword:

```
var a;
var name = "simon";
```

If you declare a variable without assigning it to anything, its value is undefined.

OPERATORS

JavaScript's numeric operators are +, -, *, / and % - which is the remainder operator. Values are assigned using =, and there are also compound assignment statements such as += and -= - these extend out to x = x operator y.

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

You can use ++ and -- to increment and decrement respectively. These can be used as prefix or postfix operators.

The + operator also does string concatenation:

```
> "hello" + " world"
hello world
```

If you add a string to a number (or other value) everything is converted in to a string first. This might catch you out:

```
> "3" + 4 + 5
345
> 3 + 4 + "5"
75
```

Adding an empty string to something is a useful way of converting it.

Comparisons in JavaScript can be made using <, >, <= and >=. These work for both strings and numbers. Equality is a little less straightforward. The double-equals operator performs type coercion if you give it different types, with sometimes interesting results:

```
> "dog" == "dog"
true
> 1 == true
true
```

To avoid type coercion, use the triple-equals operator:

```
> 1 === true
false
> true === true
true
```

There are also != and !== operators.

JavaScript also has bitwise operations. If you want to use them, they're there.

CONTROL STRUCTURES

JavaScript has a similar set of control structures to other languages in the C family. Conditional statements are supported by if and else; you can chain them together if you like:

```
var name = "kittens";
if (name == "puppies") {
  name += "!";
} else if (name == "kittens") {
  name += "!!";
} else {
  name = "!" + name;
}
name == "kittens!!"
```

JavaScript has while loops and do-while loops. The first is good for basic looping; the second for loops where you wish to ensure that the body of the loop is executed at least once:

```
while (true) {
  // an infinite loop!
}

do {
  var input = get_input();
} while (inputIsNotValid(input))
```

JavaScript's for loop is the same as that in C and Java: it lets you provide the control information for your loop on a single line.

```
for (var i = 0; i < 5; i++) {
  // Will execute 5 times
}
```

The && and || operators use short-circuit logic, which means they will execute their second operand dependant on the first. This is useful for checking for null objects before accessing their attributes:

```
var name = o && o.getName();
```

Or for setting default values:

```
var name = otherName || "default";
```

JavaScript has a tertiary operator for one-line conditional statements:

```
var allowed = (age > 18) ? "yes" : "no";
```

The switch statement can be used for multiple branches based on a number or string:

```
switch(action) {
    case 'draw':
        drawit();
        break;
    case 'eat':
        eatit();
        break;
    default:
        donothing();
}
```

If you don't add a break statement, execution will "fall through" to the next level. This is very rarely what you want - in fact it's worth specifically labelling deliberate fallthrough with a comment if you really meant it to aid debugging:

```
switch(a) {
    case 1: // fallthrough
    case 2:
        eatit();
        break;
    default:
        donothing();
}
```

The default clause is optional. You can have expressions in both the switch part and the cases if you like; comparisons take place between the two using the === operator:

```
switch(1 + 3):
    case 2 + 2:
        yay();
        break;
    default:
        neverhappens();
}
```

OBJECTS:

JavaScript objects are simply collections of name-value pairs. As such, they are similar to:

- Dictionaries in Python
- Hashes in Perl and Ruby
- Hash tables in C and C++
- HashMaps in Java
- Associative arrays in PHP

The fact that this data structure is so widely used is a testament to its versatility. Since everything (bar core types) in JavaScript is an object, any JavaScript program naturally involves a great deal of hash table lookups. It's a good thing they're so fast!

The "name" part is a JavaScript string, while the value can be any JavaScript value - including more objects. This allows you to build data structures of arbitrary complexity.

There are two basic ways to create an object:

```
var obj = new Object();
```

And;

```
var obj = {};
```

These are semantically equivalent; the second is called object literal syntax, and is more convenient. Object literal syntax was not present in very early versions of the language which is why you see so much code using the old method.

Once created, an object's properties can again be accessed in one of two ways:

```
obj.name = "Simon"
var name = obj.name;
```

And...

```
obj["name"] = "Simon";
var name = obj["name"];
```

These are also semantically equivalent. The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time. It con also be used to set and get properties with names that are reserved words:

```
obj.for = "Simon"; // Syntax error
obj["for"] = "Simon"; // works fine
```

Object literal syntax can be used to initialise an object in its entirety:

```
var obj = {
    name: "Carrot",
    "for": "Max",
    details: {
        color: "orange",
        size: 12
    }
}
```

Attribute access can be chained together:

```
> obj.details.color
orange
> obj["details"]["size"]
12
```

ARRAYS

Arrays in JavaScript are actually a special type of object - one that uses numbers instead of strings for the property names. They work very much like regular objects (always accessed using [] syntax) but they have one magic property called 'length'. This is always one more than the highest index in the array.

The old way of creating arrays is as follows:

```
> var a = new Array();
> a[0] = "dog";
> a[1] = "cat";
> a[2] = "hen";
> a.length
3
```

A more convenient notation is to use an array literal:

```
> var a = ["dog", "cat", "hen"];
> a.length
3
```

Leaving a trailing comma at the end of an array literal is inconsistent across browsers, so don't do it.

Note that array.length isn't necessarily the number of items in the array. Consider the following:

```
> var a = ["dog", "cat", "hen"];
> a[100] = "fox";
> a.length
101
```

Remember - the length of the array is one more than the highest index.

If you query a non-existent array index, you get undefined:

```
> typeof(a[90])]
undefined
```

If you take the above in to account, you can iterate over an array using the following:

```
for (var i = 0; i < a.length; i++) {
    // Do something with a[i]
}
```

This is slightly inefficient as you are looking up the length property once every loop. An improvement is this:

```
for (var i = 0, j = a.length; i < j; i++) {
    // Do something with a[i]
```

```
}
```

An even nicer idiom is:

```
for (var i = 0, item; item = a[i]; i++) {
    // Do something with item
}
```

Here we are setting up two variables. The assignment in the middle part of the for loop is also tested for truthiness - if it succeeds, the loop continues. Since i is incremented each time, items from the array will be assigned to item in sequential order. The loop stops when a falsy item is found (such as undefined).

Note that this trick should only be used for arrays which you know do not contain falsy values (arrays of objects or DOM nodes for example). If you are iterating over numeric data that might include a 0 or string data that might include the empty string you should use the i, j idiom instead.

If you want to append an item to an array, the safest way to do it is like this:

```
a[a.length] = item;
```

Since a.length is one more than the highest index, you can be assured that you are assigning to an empty position at the end of the array.

Arrays come with a number of methods:

a.toString(), a.toLocaleString(), a.concat(item, ..), a.join(sep), a.pop(), a.push(item, ..), a.reverse(), a.shift(), a.slice(start, end), a.sort(cmpfn), a.splice(start, delcount, [item]..), a.unshift([item]..)

concat returns a new array with the items added on to it.
pop removes and returns the last item
push adds one or more items to the end (like our ar[ar.length] idiom)
slice returns a sub-array
sort takes an optional comparison function
splice lets you modify an array by deleting a section and replacing it with more items
unshift prepends items to the start of the array

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't perform addition on undefined
```

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

```
> add(2, 3, 4)
5 // added the first two; 4 was ignored
```

That may seem a little silly, but 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
```

That's really not any more useful than writing 2 + 3 + 4 + 5 though. 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
```

This is pretty useful, but introduces a new problem. The avg() function takes a comma separated list of arguments - but what if you want to find the average of an array? You could just rewrite the function as follows:

```
function avgArray(arr) {
    var sum = 0;
    for (var i = 0, j = arr.length; i < j; i++) {
        sum += arguments[i];
    }
```

```
    return sum / arr.length;
}
> avgArray([2, 3, 4, 5])
3.5
```

But it would be nice to be able to reuse the function that we've already created. Luckily, JavaScript lets you call a function and call it with an arbitrary array of arguments, using the apply() method of any function object.

```
> avg.apply(null, [2, 3, 4, 5])
3.5
```

The second argument to apply() is the array to use as arguments; the first will be discussed later on. This emphasizes the fact that functions are objects too.

JavaScript lets you create anonymous functions.

```
var avg = function() {
    var sum = 0;
    for (var i = 0, j = arguments.length; i < j; i++) {
        sum += arguments[i];
    }
    return sum / arguments.length;
}
```

This is semantically equivalent to the function avg() form. It's extremely powerful, as it lets you put a full function definition anywhere that you would normally put an expression. This enables all sorts of clever tricks. Here's a way of "hiding" some local variables - like block scope in C:

```
> var a = 1;
> var b = 2;
> (function() {
    var b = 3;
    a += b;
})();
> a
4
> b
2
```

JavaScript allows you to call functions recursively. This is particularly useful for dealing with tree structures, such as you get in the browser DOM.

```
function countChars(elm) {
    if (elm.nodeType == 3) { // TEXT_NODE
        return elm.nodeValue.length;
    }
    var count = 0;
    for (var i = 0, child; child = elm.childNodes[i]; i++) {
        count += countChars(child);
    }
    return count;
}
```

This highlights a potential problem with anonymous functions: how do you call them recursively if they don't have a name? The answer lies with the arguments object, which in addition to acting as a list of arguments also provides a property called arguments.callee. This always refers to the current function, and hence can be used to make recursive calls:

```
var charsInBody = (function(elm) {
    if (elm.nodeType == 3) { // TEXT_NODE
        return elm.nodeValue.length;
    }
    var count = 0;
    for (var i = 0, child; child = elm.childNodes[i]; i++) {
        count += arguments.callee(child);
    }
    return count;
})(document.body);
```

Since arguments.callee is the current function, and all functions are abjects, you can use arguments.callee to save information across multiple calls to the same function. Here's a function that remembers how many times it has been called:

```
function counter() {
    if (!arguments.callee.count) {
        arguments.callee.count = 0;
    }
    return arguments.callee.count++;
}

> counter()
0
> counter()
1
> counter()
2
```

CUSTOM OBJECTS

In classic Object Oriented Programming, objects are collections of data and methods that operate on that data. Let's consider a person object with first and last name fields. There are two ways in which their name might be displayed: as "first last" or as "last, first". Using the functions and objects that we've discussed previously, here's one way of doing it:

```
function makePerson(first, last) {
    return {
        first: first,
        last: last
    }
}
function personFullName(person) {
    return person.first + ' ' + person.last;
}
function personFullNameReversed(person) {
    return person.last + ', ' + person.first
}
```

```
> s = makePerson("Simon", "Willison");
> personFullName(s)
Simon Willison
> personFullNameReversed(s)
Willison, Simon
```

This works, but it's pretty ugly. You end up with dozens of functions in your global namespace. What we really need is a way to attach a function to an object. Since functions are objects, this is easy:

```
function makePerson(first, last) {
    return {
        first: first,
        last: last,
        fullName: function() {
            return this.first + ' ' + this.last;
        },
        fullNameReversed: function() {
            return this.last + ', ' + this.first;
        }
    }
}
> s = makePerson("Simon", "Willison")
> s.fullName()
Simon Willison
> s.fullNameReversed()
Willison, Simon
```

There's something here we haven't seen before: the 'this' keyword. Used inside a function, 'this' refers to the current object. What that actually means is specified by the way in which you called that function. If you called it using dot notation on an object, that object becomes 'this'. If dot notation wasn't used for the call, 'this' refers to the global object. This is a frequent cause of mistakes. For example:

```
> s = makePerson("Simon", "Willison")
> var fullName = s.fullName;
> fullName()
undefined undefined
```

When we call fullName(), 'this' is bound to the global object. Since there are no global variables called first or last we get undefined for each one.

We can take advantage of the 'this' keyword to improve our makePerson function:

```
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = function() {
        return this.first + ' ' + this.last;
    }
    this.fullNameReversed = function() {
        return this.last + ', ' + this.first;
    }
}
var s = new Person("Simon", "Willison");
```

We've introduced another keyword: 'new'. New is strongly related to 'this'. What it does is it creates a brand new empty object, and then calls the function specified with 'this' set to that new object. Functions that are designed to be called by 'new' are called constructor functions. Common practise is to capitalise these functions as a reminder to call them with new.

Our person objects are getting better, but there are still some ugly edges to them. Every time we create a person object we are creating two brand new function objects within it - wouldn't it be better if this code was shared?

```
function personFullName() {
    return this.first + ' ' + this.last;
}
function personFullNameReversed() {
    return this.last + ', ' + this.first;
}
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = personFullName;
    this.fullNameReversed = personFullNameReversed;
}
```

That's better: we are creating the method functions only once, and assigning references to them inside the constructor. Can we do any better than that? The answer is yes:

```
function Person(first, last) {
    this.first = first;
    this.last = last;
}
Person.prototype.fullName = function() {
    return this.first + ' ' + this.last;
}
Person.prototype.fullNameReversed = function() {
    return this.last + ', ' + this.first;
}
```

Person.prototype is an object shared by all instances of Person. It forms part of a lookup chain: any time you attempt to access a property of Person that isn't set, JavaScript will check Person.prototype to see if that property exists there instead. As a result, anything assigned to Person.prototype becomes available to all instances of that constructor via the this object.

This is an incredibly powerful tool. JavaScript lets you modify something's prototype at any time in your program, which means you can add extra methods to existing objects at runtime:

```
> s = new Person("Simon", "Willison");
> s.firstNameCaps();
TypeError on line 1: s.firstNameCaps is not a function
> Person.prototype.firstNameCaps = function() {
    return this.first.toUpperCase()
}
> s.firstNameCaps()
```

```
SIMON
```

Interestingly, you can also add things to the prototype of built-in JavaScript objects. Let's add a method to String that returns that string in reverse:

```
> var s = "Simon";
> s.reversed()
TypeError on line 1: s.reversed is not a function
> String.prototype.reversed = function() {
    var r = '';
    for (var i = this.length - 1; i >= 0; i--) {
        r += this[i];
    }
    return r;
}
> s.reversed()
nomiS
```

Our new method even works on string literals!

```
> "This can now be reversed".reversed()
desrever eb won nac sihT
```

As I mentioned before, the prototype forms part of a chain. The root of that chain is Object.prototype, whose methods include toString() - it is this method that is called when you try to represent an object as a string. This is useful for debugging our Person objects:

```
> var s = new Person("Simon", "Willison");
> s
[object Object]
> Person.prototype.toString = function() {
    return '<Person: ' + this.fullName() + '>';
}
> s
<Person: Simon Willison>
```

Remember how avg.apply() had a null first argument? We can revisit that now. The first argument to apply() is the object that should be treated as 'this'. For example, here's a trivial implementation of 'new':

```
function trivialNew(constructor) {
    var o = {}; // Create an object
    constructor.apply(o, arguments);
    return o;
}
```

This isn't an exact replica of new as it doesn't set up the prototype chain. apply() is difficult to illustrate - it's not something you use very often, but it's useful to know about.

apply() has a sister function named call, which again lets you set 'this' but takes an expanded argument list as opposed to an array.

```
function lastNameCaps() {
```

```
    return this.last.toUpperCase();
}
var s = new Person("Simon", "Willison");
lastNameCaps.call(s);
// Is the same as:
s.lastNameCaps = lastNameCaps;
s.lastNameCaps();
```

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.

CLOSURES

This leads us to one of the most powerful abstractions that JavaScript has to offer - but also the most potentially confusing. What does this do?

```
function makeAdder(a) {
    return function(b) {
        return a + b;
    }
}
x = makeAdder(5);
y = makeAdder(20);
x(6)
?
y(7)
?
```

The name of the makeAdder function should give it away: it creates new 'adder' functions, which when called with one argument add it to the argument that they were created with.

What's happening here is pretty much the same as was happening with the inner functions earlier on: a function defined inside another function has access to the outer function's variables. The only difference here is that the outer function has returned, and hence common sense would seem to dictate that its local variables no longer exist. But they /do/ still exist - otherwise the adder functions would be unable to work. What's more, there are two different "copies" of makeAdder's local variables - one in which a is 5 and one in which a is 20.

Here's what's actually happening. When ever JavaScript executes a function, a 'scope' object is created to hold the local variables created within that function. It is initialised with any variables passed in as function parameters. This is similar to the global object that all global variables and functions live in, but with a couple of important differences: firstly, a brand new scope object is created every time a function starts executing, and secondly, unlike the global abject (which in browsers is accessible as window) these scope objects cannot be directly accessed from your JavaScript code. There is no mechanism for iterating over the properties of the current scope object for example.

So when makeAdder is called, a scope object is created with one property: a, which is the argument passed to the makeAdder function. makeAdder then returns a newly created function. Normally JavaScript's garbage collector would clean up the scope object created for makeAdder at this point, but the returned function maintains a reference back to that scope object. As a result, the scope object will not be garbage collected until there are no more references to the function object that makeAdder returned.

Scope objects form a chain called the scope chain, similar to the prototype chain used by JavaScript's object system.

A closure is the combination of a function and the scope object in which it was created.

Closures let you save state - as such, they can often be used in place of objects. NEED EXAMPLE

MEMORY LEAKS

An unfortunate side effect of closures is that they make it trivially easy to leak memory in Internet Explorer. JavaScript is a garbage collected language - objects are allocated memory upon their creation and that memory is reclaimed by the browser when no references to an object remain. Objects provided by the host environment are handled by that environment.

Browser hosts need to manage a large number of objects representing the HTML page being presented - the objects of the DOM. It is up to the browser to manage the allocation and recovery of these.

Internet Explorer uses its own garbage collection scheme for this, separate from the mechanism used by JavaScript. It is the interaction between the two that can cause memory leaks.

A memory leak in IE occurs any time a circular reference is formed between a JavaScript object and a native object. Consider the following:

```
function leakMemory() {
    var el = document.getElementById('el');
    var o = { 'el': el };
    el.o = o;
}
```

The circular reference formed above creates a memory leak; IE will not free the memory used by el and o until the browser is completely restarted.

The above case is likely to go unnoticed; memory leaks only become a real concern in long running applications or applications that leak large amounts of memory due to large data structures or leak patterns within loops.

Leaks are rarely this obvious - often the leaked data structure can have many layers of references, obscuring the circular reference.

Closures make it easy to create a memory leak without meaning to. Consider this:

```
function addHandler() {
    var el = document.getElementById('el');
    el.onclick = function() {
        this.style.backgroundColor = 'red';
    }
}
```

The above code sets up the element to turn red when it is clicked. It also creates a memory leak. Why? Because the reference to el is inadvertently caught in the closure created for the anonymous inner function. This creates a circular reference between a JavaScript object (the function) and a native object (el).

There are a number of workarounds for this problem. The simplest is this:

```
function addHandler() {
    var el = document.getElementById('el');
    el.onclick = function() {
        this.style.backgroundColor = 'red';
    }
    el = null;
}
```

This works by breaking the circular reference.

Surprisingly, one trick for breaking circular references introduced by a closure is to add another closure:

```
function addHandler() {
    var clickHandler = function() {
        this.style.backgroundColor = 'red';
    }
    (function() {
        var el = document.getElementById('el');
        el.onclick = clickHandler;
    })();
}
```

The inner function is executed straight away, and hides its contents from the closure created with clickHandler.

Another good trick for avoiding closures is breaking circular references during the window.onunload event. Many event libraries will do this for you.