JavaScript uncovered

Since 1995, JavaScript has helped transform the internet. Why and how?

Pretty much wherever you go online these days, you'll be using a web page with some level of intelligence behind it. It may be something as simple as registering the fact that you're viewing the page, it may be tracking your likes and dislikes, or it may be providing an animation to make the experience of viewing the page better and more interactive. No matter what's happening under the hood, it's most likely JavaScript code that's doing the work.

Project Mocha

Back in 1995, Netscape hired Brendan Eich, then at MicroUnity Systems Engineering, to help out with a new project called Mocha for the next version of the Netscape Navigator browser. Sun Microsystems had only recently launched Java applets – little programs that could run in a browser – but it needed some kind of 'glue code' for the browser to allow them to run. Eich decided that a simple script language would be the answer. He opted for simplicity, because he realised that in all probability it wasn't going to be programmers creating a web page with Java applets, but web designers. Those designers would be using Java applets as black boxes that they would need to easily tie into the page.

He started work on a loosely typed interpreted language that he eventually called LiveScript. Since his intended users weren't programmers, he avoided standard development niceties like compilers and a formal object-oriented system, and made the language forgiving of minor mistakes that a more formal language would signal as errors and refuse to run. He also added hooks for code written in the language to interact with the page's HTML markup so that designers could manipulate forms, images and the like.

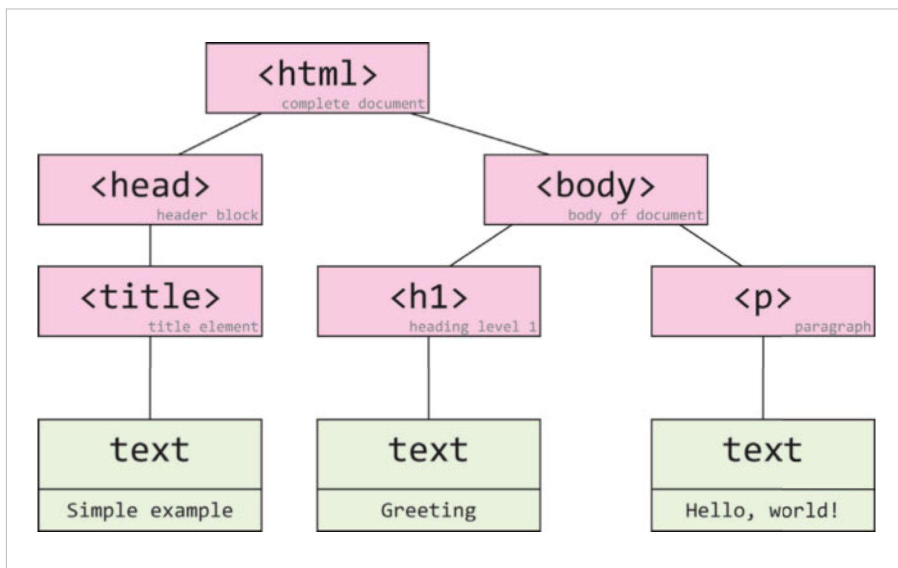
Just before Sun Microsystems and Netscape announced the new language in December 1995, it was renamed 'JavaScript' in a marketing attempt to more strongly emphasise that its purpose was to host Java applets. This was, to put it mildly, disastrous: JavaScript is not Java and, apart from some superficial syntactical resemblances, doesn't work like Java. This naming decision has led to more problems for programmers than any other, because they have the expectation that the 'Java' in JavaScript means the Java they're used to using on the server side.

The decision was also a disaster because web designers and programmers pretty much ignored Java applets, and instead used JavaScript to manipulate elements on the page (and it must be said, in the early days it was mostly image swapping: the rendering engine in the older browsers wasn't up to the task of rendering dynamic elements quickly). Since there was no need for a compiler, coupled with the fact that you could copy and paste scripts from other sites into your own with pretty much no changes, and that testing a script was so easy (just load the page, essentially) meant that JavaScript took off to an astonishing degree. 'Real' programmers dismissed it as a toy at first – the ZX81 of the web – but it soon became ubiquitous, especially once Microsoft got in on the game.

Microsoft wades in

Microsoft's response was typical of the time: it released a scripting language based on Visual Basic called VBScript. It was strongly tied to Internet Explorer, and to Windows. Since that wasn't to everyone's taste (Netscape Navigator was winning the browser wars at the time), Microsoft introduced its own version of JavaScript, called, for trademark reasons, JScript. JavaScript was a trademark of Sun Microsystems at the time, and is now owned by Oracle Corporation. The problem was that Netscape ruled the browser space, and was notorious for introducing new features and options quickly and assertively. From that era we have JavaScript itself, as well as cookies, frames and poorly thought-out HTML markup (who could forget the 'blink' attribute?). Until IE3, Microsoft was playing catch-up in the JavaScript space. ▸

Code example from w3schools.com



▲ **Figure 1: An example of a simple web page represented as a hierarchical DOM.**

Compiling JavaScript

Software companies have recognised that one of the most significant barriers to a rich web experience is JavaScript code's inability be parsed and executed quickly, without leaving the user waiting. This is especially true when loading a page: every script tag in the web page has to be parsed there, and the necessary code executed. The reason for this is that, in DHTML, you don't know what to display on the screen until the JavaScript code has been parsed and executed: it may be altering DOM elements, removing others, or even adding new elements.

To help with this, a great deal of work has been done over the past three to five years to improve the performance of JavaScript interpreters in the browser. The most impressive (in its day) example of this was Google Chrome's V8 engine: using standard benchmarks it blew away the competition. Today, all browsers have a highly optimised and tuned engine that pre-compiles JavaScript to machine code.

Eventually, with the push from IE3 users, Netscape and Sun were forced to seek the help of the European Computer Manufacturers Association (ECMA) to standardise the language, so web developers and designers would have some hope of writing code and running it without changes in the big two browsers. That led to the ECMAScript standard in 1997 (the third version of which, published in 1999, lasted for some 10 years). Yes, JavaScript is also known as ECMAScript.

Dynamic HTML

The next problem to afflict JavaScript and the browser space was Dynamic HTML (DHTML) and the Document Object Model, or DOM. When Netscape and Microsoft released version 4.0 of their respective browsers, they added better access to the attributes and elements of a web page. This access was made through a library known as the DOM, a hierarchical tree data structure that represented the web page as a set of objects, where some are siblings to others and yet others are children (or alternately, parents) to other elements.

Figure 1 shows a simple HTML document represented as a DOM tree. In this, you can see that the 'head' and 'body' elements are siblings, whereas the 'h1' element is a child of 'body' (and therefore 'body' is the parent of 'h1'). The text nodes at the bottom are also part of the DOM, albeit not exactly as HTML elements.

Unfortunately, although the general ideas about hierarchy and the DOM element names were common across IE and Navigator, Microsoft and Netscape differed on how attributes of elements were to be accessed. Although everyone was now pretty much in agreement about how the scripting language worked, using it with DHTML proved to be a nightmare – one that's still with us today.

Anyone who has ever written cross-browser JavaScript code knows the weird byroads of incompatibility between the browsers' DOM. When Internet Explorer and Netscape 4.0 came out, you could soon get libraries that tried to make Netscape act like IE, and vice versa. Other developers released libraries that encapsulated the commonality between the DOMs, though that generally meant forsaking some of the better parts of each DOM.

This split between the DOMs led to sites that worked well in one browser and less so in the other. The browser wars intensified as Microsoft, Netscape and others, together with the web standards body, the W3C, tried to standardise the DOMs. The process was lengthy, and, slowly but surely, Microsoft's Internet Explorer became the pre-eminent browser. Since IE was delivered as part of Windows, it also led to the monopoly trials against Microsoft, and the rise of the Mozilla and other browsers like Firefox and Opera.

JavaScript itself stayed pretty constant throughout this period of entrenchment and consolidation since the third edition of ECMAScript (which everyone was using) proved very stable. There were high hopes for the fourth version of JavaScript, including a 'proper' object-oriented class model, but in the end the plans proved too ambitious and they were dropped for a simpler fifth edition – ECMAScript 5 – which is being integrated into the browsers' interpreters (Firefox 4 and IE9 both support ECMAScript 5).

Cross-side scripting

Cross-side scripting (XSS) is a type of security vulnerability usually exhibited by badly-written websites. In its most common form, a hacker will inject JavaScript code intermixed with HTML into HTML forms or HTTP requests on a page in the hope that the writer of the site didn't properly validate data from these inputs and that the site will use the input as it is, unsanitised.

Although it may seem that XSS would only affect the hacker, it's possible for someone to insert this XSS attack in completely unrelated sites using frames, or to use carefully formed links to the site being attacked. With these techniques, the attacker could gain the security privileges of the user – usually without the user even being aware that the attack is happening.

What's special

JavaScript derives its syntax from the C family of languages. For example, it has the structured C-style 'if', 'while' and 'for' statements. This also means that JavaScript is an imperative language (the programmer has to explicitly state how the work is to be done, compared with a declarative language where the programmer states what the outcome is to be). As with C, statements are terminated by semicolons – although, unlike C, you can leave them off (JavaScript has a feature called automatic semicolon insertion that will attempt to place semicolons where they're needed, although it's not foolproof).

The first thing that strikes most people about JavaScript is that it's dynamically typed. This means that variables don't have any intrinsic type attached (like 'int' or 'string'). Instead it's the value that variables hold that holds the type. For example, I could define a variable x to have the value '0' (a number) and then in the next statement set it to '0' (a string).

Dynamic typing leads to a great way to use objects: duck typing. This is exemplified by the statement: 'If it quacks like a duck, walks like a duck and swims like a duck, then it can be assumed to be a duck.' So if I present to you an object that has methods called 'quack', 'walk' and 'swim', and that's all you care about, then for all intents and purposes the object is a duck, even if it looks completely different if you consider additional attributes. An example of this with JavaScript involves arrays. Arrays have some standard methods like 'shift', 'slice', 'join' and so on, which also work very well with 'array-like' objects like the arguments object.

Duck typing is also an important part of using the DOM in JavaScript: if all you know about a DOM element is that it has addEventListener (and removeEventListener)

Spotlight on… ActionScript

In the late '90s, Macromedia was trying to push its Flash product for media playing on the web. At the time, there was a limited API for interacting with the media, mostly around the idea of 'actions' like play, stop, and so on.

With Flash Player 5, Macromedia introduced a new language for interacting with Flash, known as ActionScript (**www.adobe.com/devnet/actionscript.html**). It was based on rudimentary actions of previous versions and also borrowed greatly from JavaScript, such that it was regarded as a variant of ECMAScript. ActionScript 2.0 added features like a traditional OOP class model, a type system so that you could constrain a variable to a certain type (and thereby check types at compile time). A few years later, Macromedia introduced Flex, an SDK, IDE, and a set of integration libraries.

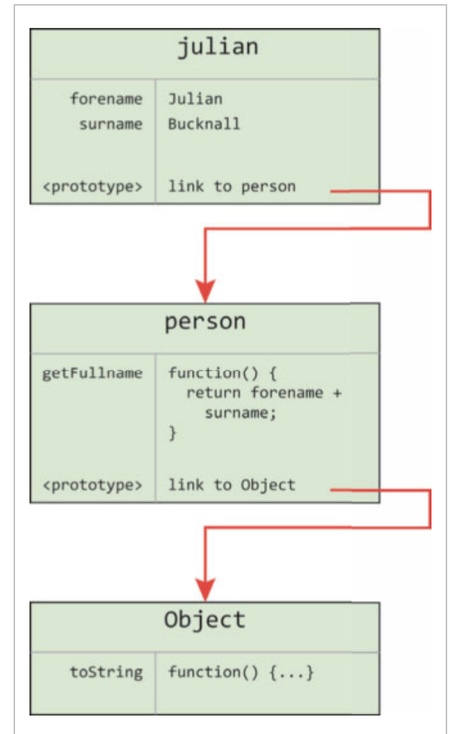
After the acquisition of Macromedia by Adobe in 2005, ActionScript 3.0 was released (**http://adobe.ly/f0aDBC**). It remains the only complete implementation of ECMAScript version 4 (the version abandoned by all JavaScript interpreters in browsers). It also adds support for packages and namespaces, and improves the performance of the class model to be more in line with prototypal inheritance. There are also several improvements that interface with Flash.

Although ActionScript was originally based on JavaScript, there are too many differences between the languages nowadays to allow any kind of interoperability. ■

then I can pass to you any object with those methods and get the observer pattern for free.

Objects are paramount in JavaScript. Apart from some primitive types like null, undefined, boolean, number and string, everything else is an object. An object in JavaScript is essentially a hash table or an associative array, that is, all properties and methods of the object are held as a table of key/value pairs. This is completely different from strongly-typed languages where an object is cast from a class template and so all such objects are pre-defined and exactly the same. In JavaScript you can add new members to an object, modify them, or even delete them. Objects have a different inheritance model than the class model we're used to with strong OOP languages: JavaScript uses prototypal inheritance where the inherited behaviour comes from a prototype object. In essence, you create an object with some desired behaviour (the prototype object) and then associate that object with a constructor function. You can then create new objects using the new keyword with the constructor, and they will all inherit their default behaviour from the prototype.

Figure 2 shows a schematic of prototypal inheritance between three objects. When you call the 'toString' method of the top 'julian' object (julian.toString()), JavaScript will first look in the 'julian' object for the method. Since it's not there, it goes to the prototype object, which is a link to the 'person' object; 'toString' isn't there either, so JavaScript follows the next prototype link to the 'ancestor' object. There it finds 'toString' and can call it. Notice that overriding a method in this scenario is accomplished by merely adding the method at a higher level – the interpreter will halt following the prototype chain earlier on.



▲ **Figure 2: A diagram of prototypal inheritance, showing how the interpreter finds the 'toString' method by following the links.**

On the server

Although most of the work in JavaScript is undoubtedly done in the browser, client-side, there's no reason why you should limit the language to that environment.

One of the more interesting developments in recent months has been the rise of 'node.js', or Node. This library was written explicitly for creating servers using JavaScript. It uses a different implementation pattern than the traditional big web server platforms (IIS and Apache), since it uses a single threaded engine. All I/O in Node, whether that is reading from or writing to a database, sending a response to a client, is done asynchronously. There is no blocking in Node waiting for a response: all activity is done by issuing a request with a callback attached. When the request in complete, the callback is called. Node can therefore process many more simultaneous requests from clients than standard server platforms can.

The downside of this is that designing and writing asynchronous code is more difficult than writing code in the traditional, blocking manner. ■

JavaScript functions

Then we come to functions in JavaScript, which are where much of the language's brilliance shines through. The first and most important feature of JavaScript functions is that they are first-class objects. You can assign them to variables. They can be passed as parameters to other functions (usually known as callbacks in this case). They can be returned from functions, so you can implement the functional programming construct known as currying. Functions that take or return other functions are known as higher-order functions.

Functions are objects, so they support their own properties and methods, and members can be inherited through the prototypal chain. So, for example, all functions have a length property (the number of parameters expected by the function) and a call method that allows you to execute the function bound to any object you like. Although a function may be defined as a method on some object, you can execute it as if it were defined on some other object.

Functions can also be nested inside other functions. When you do this, scope resolution in JavaScript happens on a function level, not at the level of blocks (like every other C-style language). That means a nested function has full access to all of the local variables defined at the outer function (and if that function is also nested, to the local variables defined in that outer function, and so on).

The next impressive thing is that the outer function can form a closure around its nested functions. In other words, a nested function can live on beyond the execution of its outer function and yet still have access to the outer function's local variables. Here's an example: imagine a function that returns another function. That returned function is a nested function. The outer function will terminate once it has return its nested function, but its local variables live on as part of a closure so that the nested function can still refer to them when it is eventually executed.

JavaScript's ability to form closures gives it much of its power, but also causes some of the difficulties involved in working with it. This feature, together with the use of higher-order functions, essentially enables JavaScript to become a functional programming language.

People use JavaScript every day, whether they know it or not when they surf the internet. Google's search auto-completion on its home page? JavaScript. Pages that reveal more information when you click an icon, like Facebook? JavaScript. Pages that refresh in real time, like Twitter? JavaScript. **PCP**

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk