**JavaScript: A Brief Introduction**

Our current digital age has been defined by far-reaching methods of design, founded with intuitive benchmarks and user-friendly interfaces. JavaScript was yet another tool that developers could use when creating their own internet content, eventually gaining notoriety as a key foundational asset in online content production. Over two decades later, JavaScript has been a pioneering force and extending what is possible with communicating over the internet, and what users can expect to find when connecting to their favorite website.

Before JavaScript could come close to its future marks, it first needed a designer. Brendan Eich, previously a MicroUnity Systems Engineering employee, was tasked by Netscape to design and implement a new language capable of allowing Netscape’s internet browser “Navigator” and its newly added Java support to be more accessible to non-Java programmers. Eich understood that thousands of web developers that had fought against a system that plagued their design process from start to finish, might find their job easier if it was not required to use a bytecode compiler or have knowledge of object-oriented software design. The term “LiveScript,” originally inspired by the language’s dynamic nature, was quickly renamed JavaScript, and took on a life of its own, “more commonly used to manipulate images and document contents than to control Java applets, perhaps reflecting the then-current trend (driven by typographer David Siegel and others) to bring full interactivity and sophisticated user interface and typography concepts to the formerly static Web,” (Champeon, 2001). With such a low barrier of entry, developers were able to take scripts and apply them to current pages in need of content without a compiler. Many found the language to be a messy, “lesser-than” language for its lack of IDE, and “simple” language limitations. Fortunately, subsequent releases of Navigator included support for script-driven interaction with plugins, a more robust security suite, and additional features aimed and enticing individuals and companies to try the language for themselves. Netscape eventually introduced a JavaScript server-side language capable of performing database queries and other advanced features, appropriately named “LiveWire,” but its success was shadowed by the popularity of ASP and Perl, and never caught on like originally planned. Microsoft attempted to respond to Netscape and Eich’s work with their own VBScript language, that was capable of hooking into embedded components but limited to Microsoft’s Windows operating system. While they attempted to keep up, Netscape’s Navigator continued to dominate its own product by beating Microsoft to the punch with newer and newer versions of JavaScript capable of image swapping and other cutting-age features for the time. Even Microsoft’s release of Internet Explorer 3.0 and its newly found support of W3C-sanctioned standards like Cascading Stylesheets, was still behind until their release of version 3.02.

As previously mentioned, JavaScript’s dynamic nature was a core principle in its design philosophy, and to this very day is a strong force to be reckoned with. Its malleable content delivery capabilities allow developers to change their code on the fly, much akin to HTML. With no IDE or compiler required, a developer could simply overwrite an old file through FTP, or edit it with a similar WYSIWYG editor on the fly, with changes to the code representing themselves virtually instantaneously. Douglas Crockford in his book, “*JavaScript: The Good Parts*,” responds to these capabilities succinctly, “Since functions are objects, they can be used like any other value. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to functions, and functions can be returned from functions. Also, since functions are objects, functions can have methods. […] A function literal can appear anywhere that an expression can appear. Functions can be defined inside of other functions. An inner function of course has access to its parameters and variables. An inner function also enjoys access to the parameters and variables of the functions it is nested within. The function object created by a function literal contains a link to that outer context. This is called closure. This is the source of enormous expressive power,” (Crockford, p. 26). The functions can further be expanded upon through invocation, and with an object oriented focus for any project in question, you can access these objects through the method itself:

// Create myObject. It has a value and an increment

// method. The increment method takes an optional

// parameter. If the argument is not a number, then 1

// is used as the default.

var myObject = {

value: 0,

increment: function (inc) {

this.value += typeof inc === 'number' ? inc : 1;

}

};

myObject.increment();

document.writeln(myObject.value); // 1

myObject.increment(2);

document.writeln(myObject.value); // 3

As seen in “JavaScript: The Definitive Guide, 4th Edition,” written by David Flanagan, the exact structure of JavaScript, as seen in the previous example, follows a few rules. It uses the Unicode character set, and unlike the 7-bit ASCII encoding, the 16-bit Unicode encoding can represent virtually any written language. Case sensitivity can produce problems for some developers not aware of such a requirement, and can cause some confusion for keywords like “while” which must be written in all lower-case letters. Whitespace and line breaks can be used freely in JavaScript programs, and indentation rarely causes syntax errors as long as simple statements are followed by semicolons (; ). “Although JavaScript theoretically allows line breaks between any two tokens, the fact that JavaScript automatically inserts semicolons for you causes some exceptions to this rule. Loosely, if you break a line of code in such a way that the line before the break appears to be a complete statement, JavaScript may think you omitted the semicolon an insert one for you, altering your meaning,” (Flanagan, p. 28).

Special kinds of objects, known as “functions” are objects that have executable code associated with it. Using Crockford’s previous example as reference, a function may be *invoked* to perform any kind of operation. Like arrays, functions behave differently and JavaScript defines special language syntax when working with them. In addition to functions and arrays, JavaScript handles a few unique types of objects. Examples include “date class”, “RegExp,” and “Error”, which define objects that represent dates, define objects that represent regular expressions, and define objects that represent syntax and runtime errors, respectively. Interestingly, JavaScript handles its numbers in base-10, and can “[…] represent all integers between 9007199254740992 (-253) and 9007199254740992 (253), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits,” (Flanagan, p. 34). These variations in input allow string, (a sequence of zero or more Unicode characters enclosed with-in quotes), must be written on a single line, and may not be broken across two lines. This same logic applies to Boolean values that represent a truth value – it explains if something is true or not.

With such a long laborious history, the evolution of JavaScript as a core scripting language for hobbyists and professional web developers alike, has allowed for a transformative scene to transcend what was previously believed to be possible. Eric Elliott, author of “*Programming JavaScript Applications*”, explains in his article titled “*The Two Pillars of JavaScript*,” “Brendan Eich didn’t invent either of the pillars, but JavaScript exposed the programming masses to them. Both pillars are equally important, but I’m concerned that a large number of JavaScript programmers are completely missing one or both innovations, because JavaScript is pretty good at letting you code poorly if you don’t bother to learn it properly. […] Even if you follow Douglas Crockford’s advice and stop using `this`, you can still do things the prototypal way. Concatenative inheritance is possible because of a feature in JavaScript known as dynamic object extension: the ability to add to an object after it has been instantiated,” (Elliott, 2014).

Ultimately, professional web developers and content creators will navigate to the easiest to code, and most effortlessly to implement coding language available. JavaScript lasting the test of time is certainly a testament to the power it awakened in a countless number of developers parched for a taste of the next big thing. Unfortunately, with the advent of mobile devices and smartphones, JavaScript has been substituted for a myriad of suite inclusions related to HTML5 and the power it brings. Numerous data breaches and security concerns over the last decade haven’t made this transitionary process any easier. Due to its high level of productivity and its low barrier of entry however, JavaScript should continue to find life in the products it helps create, and with time may even evolve into a newly adapted version capable of the unimaginable.

Works Cited

Champeon, Steve. "JavaScript: How Did We Get Here?" O'Reilly Network: Web DevCenter. O'Reilly, 06 Apr. 2001. Web. 8 July 2016.

Crockford, Doug. "Code Conventions for the JavaScript Programming Language." Code Conventions for the JavaScript Programming Language. N.p., n.d. Web. 08 July 2016.

Crockford, Douglas. JavaScript: The Good Parts. 1st ed. Beijing: O'Reilly, 2008. Print.

Elliott, Eric. "The Two Pillars of JavaScript." Medium. JavaScript Scene, 21 Oct. 2014. Web. 30 June 2016.