JavaScript Mastery

By: Andrew Gerst

http://about.me/agerst (http://about.me/agerst)
JavaScript Mastery (Source) (http://cs.unc.edu/~gerst/javascript.html)

Preface

This is a book about the JavaScript programming language. It is intended for programmers transitioning from another language such as Java as well as programmers who have been working with JavaScript at a novice level and are now ready for a more sophisticated relationship with the language.

JavaScript is a surprisingly powerful language. It's also small enough that it's easily mastered. My goal here is to help you to learn to think in JavaScript. This book isn't exhaustive about the language and won't contain everything you need to know. Instead, this book just contains the things that are really important and the things I wished someone told me when I started learning JavaScript. By the end of this book you should discover the different components of JavaScript and the ways these components can work together.

The amazing thing about JavaScript is that it is possible to get work done with it without knowing much about the language, or even knowing much about programming. It is a language with enormous expressive power. It is even better when you know what you're doing. Programming is a difficult business. It should never be undertaken in ignorance.

Here are the things I expect a front-end engineer to know without any outside help:

DOM structure - how nodes are related to one another and how to traverse from one to the next.
DOM manipulation - how to add, remove, move, copy, create, and find nodes.
Events - how to use them and the major differences between IE and the DOM event models.
XMLHttpRequest - what it is, how to perform a complete GET request, how to detect errors.
Strict vs. quirks modes - how to trigger each and why this matters.
The box model - how margin, padding, and border are related and the difference between border-box (standards mode) and content-box (old Internet Explorer) sizing.
Block vs. inline elements - how to manipulate using CSS, how they effect things around them and your ability to style them.
Floating elements - how to use them, troubles with them, and how to work around the troubles.
HTML vs. XHTML - how they're different, why you might want to use one over the other.
JSON - what it is, why you'd want to use it, how to actually use it, implementation details.

Table of Contents

What is JavaScript?

The World's Most Misunderstood Programming Language

JavaScript is a cross-platform, object-oriented scripting language.

Core JavaScript contains a core set of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects.

Netscape invented JavaScript, and JavaScript was first used in Netscape browsers.

JavaScript is a language with more than its share of bad parts. It went from nonexistence to global adoption in an alarmingly short period of time. It never had an interval in the lab when it could be tried out and polished. It went straight into Netscape Navigator 2 just as it was, and it was very rough. When Java applets failed, JavaScript became the Language of the Web by default. JavaScript's popularity is almost completely independent of its qualities as a programming language.

JavaScript is an important language because it is the language of the web browser. Its association with the browser makes it one of the most popular programming languages in the world. At the same time, it is one of the most despised programming languages in the world. The API of the browser, the Document Object Model (DOM) is quite awful, and JavaScript is unfairly blamed. The DOM would be painful to work with in any language. The DOM is poorly specified and inconsistently implemented.

The rise of JavaScript from a simple input validator to a powerful programming language could not have been predicted. JavaScript is at once a very simple and very complicated language that takes minutes to learn but years to master. To begin down the path to using JavaScript's full potential, it is important to understand its nature, history, and limitations.

Introduction & History

JavaScript (sometimes abbreviated JS) is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

JavaScript was formalized in the ECMAScript language standard and is primarily used in the form of client-side JavaScript, implemented as part of a Web browser in order to create enhanced user interfaces and dynamic websites.

This enables programmatic access to computational objects within a host environment.

JavaScript's use in applications outside Web pages--for example in PDF documents, site-specific browsers, and desktop widgets--is also significant. Newer and faster JavaScript VMs and frameworks built upon them (notably Node.js) have also increased the popularity of JavaScript for server-side web applications.

JavaScript uses syntax influenced by that of C. JavaScript copies many names and naming conventions from Java, but the two languages are otherwise unrelated and have very different semantics. The key design principles within JavaScript are taken from the Self and Scheme programming languages.

Very Brief History

Ten days in May 1995 "Mocha"
September 1995: "LiveScript"
December 1995: "JavaScript"
1996-1997: ECMA-262 Ed. 1 (http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf), aka ES1
1999: ES3: modern JS baseline
2005: the Ajax revolution
2008: ES4 RIP, Harmony founded in July
2009: ES5: "use strict", JSON, Object.create, etc.
2012: ES6 under way: modules, let, proxies, etc.

Birth at Netscape

JavaScript was originally developed in Netscape, by Brendan Eich. Battling with Microsoft over the Internet, Netscape considered their client-server solution as a distributed OS, running a portable version of Sun Microsystems' Java. Because Java was a competitor of C++ and aimed at professional programmers, Netscape also wanted a lightweight interpreted language that would complement Java by appealing to nonprofessional programmers, like Microsoft's Visual Basic.

Developed under the name Mocha, LiveScript was the official name for the language when it first shipped in beta releases of Netscape Navigator 2.0 in September 1995, but it was renamed JavaScript when it was deployed in the Netscape browser version 2.0B3.

The change of name from LiveScript to JavaScript roughly coincided with Netscape adding support for Java technology in its Netscape Navigator web browser. The final choice of name caused confusion, giving the impression that the language was a spin-off of the Java programming language, and the choice has been characterized by many as a marketing ploy by Netscape to give JavaScript the cachet of what was then the hot new web programming language. It has also been claimed that the language's name is the result of a co-marketing deal between Netscape and Sun, in exchange for Netscape bundling Sun's Java runtime with its then-dominant browser.

Contrary to what the name suggests, JavaScript has very little to do with the programming language named Java. The similar name was inspired by marketing considerations, rather than good judgement. In 1995, when JavaScript was introduced by Netscape, the Java language was being heavily marketed and was gaining in popularity. Apparantely, someone thought it a good idea to try and ride along on this success. Now we are stuck with the name.

Server-side JavaScript

Netscape introduced an implementation of the language for server-side scripting with Netscape Enterprise Server, first released in December, 1994 (soon after releasing JavaScript for browsers). Since the mid-2000s, there has been a proliferation of server-side JavaScript implementations. Node.js is one recent notable example of server-side JavaScript being used in real-world applications.

Adoption by Microsoft

JavaScript very quickly gained widespread success as a client-side scripting language for web pages. Microsoft introduced JavaScript support in its own web browser, Internet Explorer, in version 3.0, released in August 1996. Microsoft's webserver, Internet Information Server, introduced support for server-side scripting in JavaScript with release 3.0 (1996). Microsoft started to promote webpage scripting using the umbrella term Dynamic HTML.

Microsoft's JavaScript implementation was later renamed JScript to avoid trademark issues. JScript added new date methods to fix the Y2K-problematic methods in JavaScript, which were based on Java's java.util.Date class.

Microsoft did not want to deal with Sun about the trademark issue, and so they called their implementation JScript. A lot of people think that JScript and JavaScript are different but similar languages. That's not the case. They are just different names for the same language, and the reason the names are different was to get around trademark issues.

Standardization

In November 1996, Netscape announced that it had submitted JavaScript to Ecma International for consideration as an industry standard, and subsequent work resulted in the standardized version named ECMAScript.

Later developments

JavaScript has become one of the most popular programming languages on the web. Initially, however, many professional programmers denigrated the language because its target audience was web authors and other such "amateurs", among other reasons. The advent of Ajax returned JavaScript to the spotlight and brought more professional programming attention. The result was a proliferation of comprehensive frameworks and libraries, improved JavaScript programming practices, and increased usage of JavaScript outside of web browsers, as seen by the proliferation of server-side JavaScript platforms.

In January 2009, the CommonJS project was founded with the goal of specifying a common standard library mainly for JavaScript development outside the browser.

Trademark

Today, "JavaScript" is a trademark of Oracle Corporation. It is used under license for technology invented and implemented by Netscape Communications and current entities such as the Mozilla Foundation.

ECMAScript

Related to JavaScript is a thing called ECMAScript. When browsers other than Netscape started to support JavaScript, or something that resembled it, a document was written to describe precisely how the language should work. The language described in this document is called ECMAScript, after the organization that standardized it.

ECMAScript describes a general-purpose programming language, and does not say anything about the integration of this language in a web browser. JavaScript is ECMAScript plus extra tools for dealing with Internet pages and browser windows.

A few other pieces of software use the language described in the ECMAScript document. Most importantly, the ActionScript language used by Flash is based on ECMAScript (though it does not precisely follow the standard). Flash is a system for adding things that move and make lots of noise to web-pages. Knowing JavaScript won't hurt if you ever find yourself learning to build Flash movies.

There are five editions of ECMA-262 (http://www.ecma-international.org/publications/standards/Ecma-262-arch.htm) published. ECMAScript Edition 1 (http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf) was based on several originating technologies, the most well known being

JavaScript™ (Netscape Communications) and JScript™ (Microsoft Corporation). The development of this Standard was started in November 1996. The ECMA Standard was adopted by the ECMA General Assembly of June 1997. Please note that for ECMAScript Edition 4 the Ecma standard number "ECMA-262 Edition 4" was reserved but not used in the Ecma publication process. Therefore "ECMA-262 Edition 4" as an Ecma International publication does not exist. Work on a future edition, codenamed "Harmony", is in progress.

JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in some others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript follows most Java expression syntax, naming conventions and basic control-flow constructs which was the reason why it was renamed from LiveScript to JavaScript.

In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a small number of data types representing numeric, Boolean, and string values. JavaScript has a prototype-based object model instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports functions without any special declarative requirements. Functions can be properties of objects, executing as loosely typed methods.

JavaScript is a very free-form language compared to Java. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to implement interfaces. Variables, parameters, and function return types are not explicitly typed.

Java is a class-based programming language designed for fast execution and type safety. Type safety means, for instance, that you can't cast a Java integer into an object reference or access private memory by corrupting Java bytecodes. Java's class-based model means that programs consist exclusively of classes and their methods. Java's class inheritance and strong typing generally require tightly coupled object hierarchies. These requirements make Java programming more complex than JavaScript programming.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically typed languages such as HyperTalk and dBASE. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

A common misconception is that JavaScript is similar or closely related to Java. It is true that both have a C-like syntax, the C language being their most immediate common ancestor language. They are both object-oriented, typically sandboxed (when used inside a browser), and are widely used in client-side Web applications. In addition, JavaScript was designed with Java's syntax and standard library in mind. In particular, all Java keywords were reserved in original JavaScript, JavaScript's standard library follows Java's naming conventions, and JavaScript's Math and Date objects are based on classes from Java 1.0.

"JS had to 'look like Java' only less so, [it had to] be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JS would have happened."
--Brendan Eich

However, the similarities end there. Java has static typing; JavaScript's typing is dynamic (meaning a variable can hold an object of any type and cannot be restricted). JavaScript is weakly typed (`'0.0000' == 0`, `0 == ""`, `false == ""`, etc.) while Java is more strongly typed. Java is loaded from compiled bytecode; JavaScript is loaded as human-readable source code. Java's objects are class-based; JavaScript's are prototype-based. JavaScript also has many functional programming features based on the Scheme language.

Hello World

There is no built-in I/O functionality in JavaScript; the runtime environment provides that. Here are a couple methods of producing output in the browser.

```
/**
* Hello World
*/

alert("Hello World!");
window.alert("Hello World!");
window['alert']("Hello World!");
console.log("Hello World!");
document.write("Hello World!");
document.writeln("Hello World!");
document.body.innerHTML = "Hello World!";
document.body.appendChild(document.createTextNode("Hello World!"));
document.documentElement.lastChild.appendChild(document.createTextNode("Hello World!"));
```

alert

An alert box is often used if you want to make sure information comes through to the user. When an alert box pops up, the user will have to click "OK" to proceed.

The alert dialog should be used for messages which do not require any response on the part of the user, other than the acknowledgement of the message. Dialog boxes are modal windows - they prevent the user from accessing the rest of the program's interface until the dialog box is closed. For this reason, you should not overuse any function that creates a dialog box (or modal window).

alert() is a method of the window object that cannot interpret HTML tags.

alert() must output a string, so it must first convert the value to string - it is very important if you want to check what value you have at some point and you picked alert() to fulfill that task. It also stops the execution of the script (this may be useful sometimes).

- blocks the entire browser and javascript thread
- only prints strings
- requires user interaction to continue (this means you can't automate browser usage)
- blocked by common popup blockers
- doesn't work in non-browser environments like node.js (however console.log does work in node.js)

```
alert(
    'This is an alert with basic formatting\n\n'
    + '\t• list item 1\n'
    + '\t• list item 2\n'
    + '\t• list item 3\n\n'
    + '▬▬▬▬▬▬▬๑۩۩๑▬▬▬▬▬▬▬\n\n'
    + 'Simple table\n\n'
    + 'Char\t| Result\n'
    + '\\n\t| line break\n'
    + '\\t\t| tab space'
);
```

Click here to see an alert with basic formatting

document.write

The document.write methods outputs a string directly into page.

This is one of most ancient methods of appending to a document. It is very rarely used in modern web applications, but still it has it's special place.

When document is loading, a script may document.write(text) into the document. The text will be rendered same way as if it were in HTML.

There are no restrictions on the contents of document.write. It doesn't have to output valid tags, close them or anything.

It is very fast, because the browser doesn't have to modify an existing DOM structure.

Sometimes the scripts are added by the document.write. Don't use this method, as the rest of the page will await for script loading and execution.

There is also a variation named document.writeln(text) which appends '\n' after the text.

When the page finishes loading, the document becomes closed. An attempt to document.write in it will cause the contents to be erased.

Features

The following features are common to all conforming ECMAScript implementations, unless explicitly specified otherwise.

Imperative and structured

JavaScript supports much of the structured programming syntax from C (e.g., if statements, while loops, switch statements, etc.). One partial exception is scoping: C-style block-level scoping is not supported (instead, JavaScript has function-level scoping). JavaScript 1.7, however, supports block-level scoping with the let keyword. Like C, JavaScript makes a distinction between expressions and statements. One syntactic difference from C is automatic semicolon insertion, in which the semicolons that terminate statements can be omitted.

Dynamic

dynamic typing
As in most scripting languages, types are associated with values, not with variables. For example, a variable x could be bound to a number, then later rebound to a string. JavaScript supports various ways to test the type of an object, including duck typing.

object based
JavaScript is almost entirely object-based. JavaScript objects are associative arrays, augmented with prototypes (see below). Object property names are string keys: obj.x = 10 and obj['x'] = 10 are equivalent, the dot notation being syntactic sugar. Properties and their values can be added, changed, or deleted at run-time. Most properties of an object (and those on its prototype inheritance chain) can be enumerated using a for...in loop. JavaScript has a small number of built-in objects such as Function and Date.

run-time evaluation
JavaScript includes an eval function that can execute statements provided as strings at run-time.

Functional

first-class functions
Functions are first-class; they are objects themselves. As such, they have properties and methods, such as .call()

and .bind(); and they can be assigned to variables, passed as arguments, returned by other functions, and manipulated like any other object. Any reference to a function allows it to be invoked using the () operator.

nested functions and closures
"Inner" or "nested" functions are functions defined within another function. They are created each time the outer function is invoked. In addition to that, each created function forms a lexical closure: the lexical scope of the outer function, including any constants, local variables and argument values, become part of the internal state of each inner function object, even after execution of the outer function concludes.

Prototype-based

prototypes
JavaScript uses prototypes instead of classes for inheritance. It is possible to simulate many class-based features with prototypes in JavaScript.

functions as object constructors
Functions double as object constructors along with their typical role. Prefixing a function call with new creates a new object and calls that function with its local this keyword bound to that object for that invocation. The constructor's prototype property determines the object used for the new object's internal prototype. JavaScript's built-in constructors, such as Array and String, also have prototypes that can be modified.

functions as methods
Unlike many object-oriented languages, there is no distinction between a function definition and a method definition. Rather, the distinction occurs during function calling; a function can be called as a method. When a function is called as a method of an object, the function's local this keyword is bound to that object for that invocation.

Miscellaneous

run-time environment
JavaScript typically relies on a run-time environment (e.g. in a web browser) to provide objects and methods by which scripts can interact with "the outside world". In fact, it relies on the environment to provide the ability to include/import scripts (e.g. HTML <script> elements). (This is not a language feature per se, but it is common in most JavaScript implementations.)

variadic functions
An indefinite number of parameters can be passed to a function. The function can access them through formal parameters and also through the local arguments object.

array and object literals
Like many scripting languages, arrays and objects (associative arrays in other languages) can each be created with a succinct shortcut syntax. In fact, these literals form the basis of the JSON data format.

regular expressions
JavaScript also supports regular expressions in a manner similar to Perl, which provide a concise and powerful syntax for text manipulation that is more sophisticated than the built-in string functions.

Object Everything

In JavaScript, almost everything is an object, and an object is just a hashmap. All primitive types except null and undefined are treated as objects. They can be assigned properties (assigned properties of some types are not persistent), and they have all characteristics of objects.

All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor.

```
var myObj = new Object(),
    str = "myString",
    rand = Math.random(),
    obj = new Object();

myObj.type = "Dot syntax";
myObj["date created"] = "String with space";
myObj[str] = "String value";
myObj[rand] = "Random Number";
myObj[obj] = "Object";
myObj[""] = "Even an empty string";

// myObj[obj] can be referenced by myObj[new Object()], myObj[{}], as well as myObj["[object Object]"]
```

```
myObj -> Array
(
    [type] => Dot syntax
    [date created] => String with space
    [myString] => String value
    [0.06643031584098935] => Random Number
    [[object Object]] => Object
    [] => Even an empty string
)
```

Trying Programs

Now that you've been introduced to the language I'd like to provide a couple methods so you can try JavaScript yourself.

http://jsfiddle.net/ (http://jsfiddle.net/)
http://jsbin.com/ (http://jsbin.com/)
http://livecoding.io/ (http://livecoding.io/)
https://compilr.com/ (https://compilr.com/)

Minify And Beautify JavaScript Code

http://dean.edwards.name/packer/ (http://dean.edwards.name/packer/)
http://jsbeautifier.org/ (http://jsbeautifier.org/)

Another approach is to simply create an HTML file containing the program and load it in your browser. For example, you could create a file called *test.html* with the following content:

```
<html><body><script type="text/javascript">

document.write("Hello World!");

</script></body></html>
```

Hosting Environments

XAMPP (http://www.apachefriends.org/en/xampp.html)
Heroku (https://www.heroku.com/)
000webhost (http://www.000webhost.com/)
Webs (http://www.webs.com/)

Google Sites (https://sites.google.com/)

Testing Script Performance

Sometimes after all day long coding your code becomes not so effective and your code (usually interface related) becomes slow. You have done so many changes and don't exactly know what is slowing it down. In cases like this (and of course, plenty other cases) you can test your JavaScript code performance.

A widely used threshold of time is 100 ms for when people start to notice latency in applications.

Console can be used for logging or printing out debugging information to the console. Console also has one handy method for tracking time in milliseconds.

You can use this script to test your JavaScript code. timerName in the code can be any name for your timer. Don't forget to end your timer using the same name for timeEnd().

```
console.time('timerName');

// code goes here

console.timeEnd('timerName');
```

There are also sites available online that have many different test case examples for JavaScript performance across multiple different browsers.

http://jsperf.com/ (http://jsperf.com/)

Google Web Toolkit has a browser extension called Speed Tracer, it's a tool to help you identify and fix performance problems in your web applications.

Using Speed Tracer you are able to get a better picture of where time is being spent in your application. This includes problems caused by JavaScript parsing and execution, layout, CSS style recalculation and selector matching, DOM event handling, network resource loading, timer fires, XMLHttpRequest callbacks, painting, and more.

https://developers.google.com/web-toolkit/speedtracer/ (https://developers.google.com/web-toolkit/speedtracer/)

JSLitmus is a lightweight tool for creating ad-hoc JavaScript benchmark tests.

http://www.broofa.com/Tools/JSLitmus/ (http://www.broofa.com/Tools/JSLitmus/)

This is how you examine the performance between function expression and function constructor using JSLitmus:

```
JSLitmus.test('new Function ... ', function(){
    return new Function('for(var i=0; i<100; i++) {}');
});

JSLitmus.test("function() ...", function(){
    return (function(){ for(var i=0; i<100; i++) {} });
});
```

Run a free website speed test from multiple locations around the globe using real browsers (IE and Chrome) and at real consumer connection speeds. You can run simple tests or perform advanced testing including multi-step transactions, video capture, content blocking and much more. Your results will provide rich diagnostic information including resource loading waterfall charts, Page Speed optimization checks and suggestions for improvements.

http://www.webpagetest.org/ (http://www.webpagetest.org/)

Successful societies and institutions recognize the need to record their history - this provides a way to review the past, find explanations for current behavior, and spot emerging trends. In 1996 Brewster Kahle realized the cultural significance of the Internet and the need to record its history. As a result he founded the Internet Archive which collects and permanently stores the Web's digitized content.

In addition to the content of web pages, it's important to record how this digitized content is constructed and served. The HTTP Archive provides this record. It is a permanent repository of web performance information such as size of pages, failed requests, and technologies utilized. This performance information allows us to see trends in how the Web is built and provides a common data set from which to conduct web performance research.

http://httparchive.org/ (http://httparchive.org/)

We can always measure time taken by any function by simple date object. Here I show a function used to calculate the time elapsed while a given function is executed.

```
var start = +new Date(); // log start timestamp
fn();
var end = +new Date(); // log end timestamp
var diff = end - start;

var perf = function(testName, fn){
    var startTime = new Date().getTime();
    fn();
    var endTime = new Date().getTime();
    console.log(testName + ": " + (endTime - startTime) + "ms");
};
```

JavaScript works with the number of milliseconds since the epoch (1 January 1970 00:00:00 UTC) whereas most other languages work with the seconds. You could work with milliseconds but as soon as you pass a value to say PHP, the PHP native functions will probably fail. So to be sure it's always a good idea to use the seconds, not milliseconds.

To get the Unix timestamp such as the one returned by PHP time() function, divide this number by 1000, round or floor if necessary.

Bitwise OR | 0 is similar to Math.floor() since it is a bit operation (that does not work with floats). usually its even faster than Math.floor() since it is not a function call, it is a native javascript operator.

All bitwise operations except unsigned right shift, >>>, work on signed 32-bit integers. So using bitwise operations will convert a float to an integer. This is truncation as opposed to flooring.

Disadvantage: it only works up to 2^31-1 which is around 2 billion (10^9). The max Number value is around 10^308.

Here is a jsperf test comparing Floor with Bitwise OR.

http://jsperf.com/or-vs-floor/2 (http://jsperf.com/or-vs-floor/2)

```
var unixDate = {};
unixDate["Math.round(+new Date() / 1000)"] = Math.round(+new Date() / 1000);
unixDate["Math.floor((new Date()).getTime() / 1000)"] = Math.floor((new Date()).getTime() / 1000);
unixDate["Date.now() / 1000 | 0"] = Date.now() / 1000 | 0;
```

```
unixDate -> Array
(
    [Math.round(+new Date() / 1000)] => 1366877297
```

```
    [Math.floor((new Date()).getTime() / 1000)] => 1366877296
    [Date.now() / 1000 | 0] => 1366877296
)
```

Here are varous methods of making the Date.now() function available on all browsers since Date.now() is from JavaScript 1.5, and is not supported on IE 8.

```
var $time = Date.now || function(){
    return +new Date;
};

$time();

// OR

if (typeof Date.now == "undefined") {
    Date.now = function(){return new Date().getTime()};
}
```

The various methods used to get the current timestamp (number of milliseconds since the epoch) include:

```
var myDate = new Object();
myDate["Date.now()"] = Date.now();
myDate["+new Date()"] = +new Date();
myDate["new Date().getTime()"] = new Date().getTime();
myDate["Number(new Date())"] = Number(new Date());
myDate["new Date().valueOf()"] = new Date().valueOf();
```

```
myDate -> Array
(
    [Date.now()] => 1366877296608
    [+new Date()] => 1366877296608
    [new Date().getTime()] => 1366877296608
    [Number(new Date())] => 1366877296608
    [new Date().valueOf()] => 1366877296608
)
```

Here is a link to a jsperf test comparing Date.now(), (new Date()).getTime(), and +new Date(). As you can see the (new Date()) has parentheses they are optional for object constructors, they are superfluous.

http://jsperf.com/date-now-vs-new-date-gettime/4 (http://jsperf.com/date-now-vs-new-date-gettime/4)

Date.now() is the fastest while +new Date() is the slowest.

The method performance.now() is quickly getting implemented across browsers. performance.now() is a measurement of floating point milliseconds since that particular page started to load.

There are a few situations where you'd use this high resolution timer instead of grabbing a basic timestamp:
- benchmarking
- game or animation runloop code
- calculating framerate with precision
- cueing actions or audio to occur at specific points in an animation or other time-based sequence

The high resolution timer is currently available in Chrome (Stable) as window.performance.webkitNow(), and this value is generally equal to the new argument value passed into the requestAnimationFrame callback. Pretty soon,

WebKit will drop its prefix and this will be available through performance.now(). The WebPerfWG in particular, led by Jatinder Mann of Microsoft, has been very successful in unprefixing their features quite quickly.

In summary, performance.now() is...
- a double with microseconds in the fractional
- relative to the navigationStart of the page rather than to the UNIX epoch
- not skewed when the system time changes

JavaScript Debugging

A debugging tool is essential for JavaScript development. Firefox provides a debugger via the Firebug extension; Safari and Chrome provide built-in consoles.

Each console offers:
- single- and multi-line editors for experimenting with JavaScript
- an inspector for looking at the generated source of your page
- a Network or Resources view, to examine network requests

When you are writing JavaScript code, you can use the following methods to send messages to the console:

console.log() for sending general log messages
console.dir() for logging a browseable object
console.warn() for logging warnings
console.error() for logging error messages

Other console methods are also available, though they may differ from one browser to another. The consoles also provide the ability to set break points and watch expressions in your code for debugging purposes.

Typing *$0* in Chrome will log the currently selected element.
Typing *$1* in Chrome will log the previously selected element and so forth.
Typing *$n* in Chrome will log the n^{th} previously selected element.

Functions

Different methods of invoking a function declaration. Even though in some browsers using void and delete can lead to faster results it is still recommended to use the parentheses or a bang because they are programming conventions that indicate the function is going to be invoked. JS will not allow you to fire off a function declaration (because of a concept of "variable hoisting".

http://jsperf.com/bang-function (http://jsperf.com/bang-function)
http://jsperf.com/self-invoking-function (http://jsperf.com/self-invoking-function)

Self-Invoking Functions

Ideally you should be able to do all this simply as:

```
function(){
    // do stuff
}();
```

That means declare anonymous function and execute it. But that will not work due to specifics of JS grammar.

So shortest form of achieving this is to use some expression e.g. UnaryExpression (and so CallExpression):

```
(function(){})();
(function(){}());
!function(){}();
-function(){}();
+function(){}();
~function(){}();
void function(){}();
delete function(){}();
```

The opening perenthesis and all of the other operators change the statement from a function declaration to a function expression.

In Javascript, a line beginning with function is expected to be a function statement and is supposed to look like:

```
function doSomething(){}
```

A self-invoking function like:

```
function(){}();
```

doesn't fit that form (and will cause a syntax error at the first opening paren because there is no function name), so the parens are used to delineate an anonymous function expression.

But anything that creates an expression (as opposed to a function statement) will do, so hence the !. It's telling the interpreter that this is not a function statement. Other than that, operator precedence dictates that the function is invoked before the negation.

Anybody reading the !function at the top of a large block of code will expect a self-invocation, the way we are conditioned already to expect the same when we see (function. Except that we will lose those annoying parentheses. I would expect that's the reason, as opposed to any savings in speed or character count. The syntax with the ! is useful if you write javascript without semicolons.

Inside and outside parens are identical unless you add new beforehand and .something afterwards.

Code 1:

```
new (function(){
    this.prop = 4;
}) ().prop;
```

This code creates a new instance of this function's class, then gets the prop property of the new instance.

It returns 4.

It's equivalent to:

```
function MyClass(){
    this.prop = 4;
}
```

```
new MyClass().prop;
```

Code 2:

```
new (function(){
    return { Class: function(){} };
}()).Class;
```

This code calls new on the Class property.

Since the parentheses for the function call are inside the outer set of parentheses, they aren't picked up by the new expression, and instead call the function normally, returning its return value.

The new expression parses up to the .Class and instantiates that. (the parentheses after new are optional).

It's equivalent to:

```
var namespace = { Class: function(){} };

function getNamespace(){ return namespace; }

new (getNamespace()).Class;
// OR
new namespace.Class;
```

Without the parentheses around the call to getNamespace(), this would be parsed as (new getNamespace()).Class – it would call instantiate the getNamespace class and return the Class property of the new instance.

void Operator

The void operator takes a single operand, evaluates the given expression, and then returns undefined. This is not useful, and it is very confusing. Avoid void.

Uses

This operator allows inserting expressions that produce side effects into places where an expression that evaluates to undefined is desired.

The void operator is often used merely to obtain the undefined primitive value, usually using "void(0)" (which is equivalent to "void 0"). In these cases, the global variable undefined can be used instead (assuming it has not been assigned to a non-default value).

JavaScript URIs

When a browser follows a javascript: URI, it evaluates the code in the URI and then replaces the contents of the page with the returned value, unless the returned value is undefined. The void operator can be used to return undefined. For example:

```
<a href="javascript:void(0);">Click here to do nothing</a>
<a href="javascript:void(document.body.style.backgroundColor='green');">Click here for green background</a>
```

Note, however, that the javascript: pseudo protocol is discouraged over other alternatives, such as unobtrusive event handlers.

delete operator

The delete operator removes a property from an object.

Syntax

delete expression

where expression should evaluate to a property reference, e.g.:

```
delete object.property
delete object['property']
delete object[index]
delete property // deletes properties of the global object, or, using the with statement, properties of the referenced object
```

If expression does not evaluate to a property, delete does nothing.

Parameters

object: The name of an object, or an expression evaluating to an object.
property: The property to delete.
index: An integer representing the array index to delete.

Returns

Returns false only if the property exists on the object itself, regardless of its prototypes, and cannot be deleted. It returns true in all other cases.

Description

If the delete operator succeeds, it removes the property from the object entirely. However, if a property with the same name exists on the object's prototype chain, the object will inherit that property from the prototype.

delete is only effective on an object's properties. It has no effect on variable or function names.
While sometimes mischaracterized as global variables, assignments that don't specify an object (e.g. x = 5) are actually property assignments on the global object.

delete can't remove certain properties of predefined objects (like Object, Array, Math etc). These are marked in the ECMA 262 specification as DontDelete.

```
x = 42;         // creates the property x on the global object
var y = 43;     // declares a new variable, y
myobj = {};
myobj.h = 4;    // creates property h on myobj
myobj.k = 5;    // creates property k on myobj

delete x;       // returns true  (x is a property of the global object and can be deleted)
delete y;       // returns false (delete doesn't affect variable names)
delete Math.PI; // returns false (delete doesn't affect certain predefined properties)
delete myobj.h; // returns true  (user-defined properties can be deleted)

with (myobj) {
```

```
    delete k;   // returns true  (equivalent to delete myobj.k)
}

delete myobj;   // returns true  (myobj is a property of the global object, not a variable, so it can be deleted).
```

If the object inherits a property from a prototype, and doesn't have the property itself, the property can't be deleted by referencing the object. You can, however, delete it directly on the prototype.

```
function Foo(){}
Foo.prototype.bar = 42;
var foo = new Foo();
delete foo.bar;           // returns true, but with no effect, since bar is an inherited property
alert(foo.bar);           // alerts 42, property still inherited
delete Foo.prototype.bar; // deletes property on prototype
alert(foo.bar);           // alerts "undefined", property no longer inherited
Deleting array elements
```

When you delete an array element, the array length is not affected. For example, if you delete a[3], a[4] is still a[4] and a[3] is undefined. This holds even if you delete the last element of the array (delete a[a.length-1]).

When the delete operator removes an array element, that element is no longer in the array. In the following example, trees[3] is removed with delete.

```
var trees = ["redwood","bay","cedar","oak","maple"];

delete trees[3];
if (3 in trees) {
    // this does not get executed
}
```

If you want an array element to exist but have an undefined value, use the undefined value instead of the delete operator. In the following example, trees[3] is assigned the value undefined, but the array element still exists:

```
var trees = ["redwood","bay","cedar","oak","maple"];
trees[3] = undefined;
if (3 in trees) {
    // this gets executed
}
```

Reserved Words

JavaScript has a number of "reserved words," or words that have special meaning in the language. You should avoid using these words in your code except when using them with their intended meaning.

```
abstract
boolean
break
byte
case
catch
char
class
```

```
const
continue
debugger
default
delete
do
double
else
enum
export
extends
final
finally
float
for
function
goto
if
implements
import
in
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
super
switch
synchronized
this
throw
throws
transient
try
typeof
var
void
volatile
while
with
```

parseInt function

```
ECMAScript 5 Removes Octal Interpretation

The ECMAScript 5 specification of the function parseInt no longer allows implementations to treat Strings
beginning with a 0 character as octal values. ECMAScript 5 states:

The parseInt function produces an integer value dictated by interpretation of the contents of the string argument
according to the specified radix. Leading white space in string is ignored. If radix is undefined or 0, it is
assumed to be 10 except when the number begins with the character pairs 0x or 0X, in which case a radix of 16 is
```

assumed. If radix is 16, number may also optionally begin with the character pairs 0x or 0X.

This differs from ECMAScript 3 (http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf), which discouraged but allowed octal interpretation.

Since many implementations have not adopted this behavior as of 2011, and because older browsers must be supported, always specify a radix.

First of all, you really don't need parseInt() in most cases. It's algorithm is full of various quirks, the 0 prefix is even forbidden by the specification ("the specification of the function parseInt no longer allows implementations to treat Strings beginning with a 0 character as octal values."), but it will take a while to change browser behaviors (even if I'm sure that nobody does use octals intentionally in parseInt()). And Internet Explorer 6 will never change (the Internet Explorer 9 however removed support for octals in parseInt()). The algorithm used by it usually does more than you want from it. In certain cases, it's bad idea.

1. First argument is converted to string if it isn't already.
2. Trim the number, so ' 4' becomes '4'.
3. Check if string begins with - or + and remove this character. If it was - make output negative.
4. Convert radix to integer.
5. If radix is 0 or NaN try to guess radix. It means looking (case-insensitive) for 0x and (non-standard) 0. If prefix wasn't found, 10 is used (and this is what you most likely what).
6. If radix is 16 strip 0x from the beginning if it exists.
7. Find the first character which is not in range of radix.
8. If there is nothing before first character which wasn't in range of radix, return NaN.
9. Convert number to decimal until the first character which is not in range.

For example, parseInt('012z', 27) gives 0 * Math.pow(27, 2) + 1 * Math.pow(27, 1) + 2 * Math.pow(27, 0).

The algorithm itself is not really quick, but performance varies (optimizations make wonders). I've put test on JSPerf and the results are interesting. + and ~~ are fastest with exception for Chrome where parseFloat() is somehow way faster than other options (2 to 5 times faster than other options, where + is actually 5 times slower). In Firefox, ~~ test is very fast - in certain cases, I've got Infinity cycles.

The other thing is correctness. parseInt(), ~~ and parseFloat() make errors silent. In case of parseInt() and parseFloat() characters are ignored after invalid character - you can call it a feature (in most cases it's anti-feature for me, just like switch statements fallthrough) and if you need it, use one of those. In case of ~~ it means returning 0, so be careful.

In certain cases, parseInt() might hurt you. Badly. For example, if number is so big that it is written in exponential notation. Use Math methods then.

parseInt(2e30); // will return 2

Anyways, at end I want to make a list when of methods to convert strings to numbers (both integers and floats). They have various usages and you may be interested what method to use. In most cases, the simplest one is +number method, use it if you can. Whatever you do (except for first method), all should give correct result.

```
parseInt('08', 10); // 8
+'08';              // 8
~~'08';             // 8
parseFloat('08');   // 8
Number('08');       // 8
new Number('08');   // 8... I meant Object container for 8
Math.ceil('08');    // 8
```

Don't use parseInt(number). Simple as that. Either use parseInt(number, 10) or this workaround which will magically fix parseInt function. Please note that this workaround will not work in JSLint. Please don't complain about it.

```
(function(){
    "use strict";
    var oldParseInt = parseInt;
    // Don't use function parseInt(){}. It will make local variable.
    parseInt = function(number, radix){
        return oldParseInt(number, radix || 10);
    };
}());

parseInt(number, radix);
```

parseInt converts argument to numbers using mentioned above algorithm. Avoid using it on large integers as it can do incorrect results in cases like parseInt(2e30). Also, never ever give it as argument to Array.prototype.map or Underscore.js variation of it as you may get weird results (try ['1', '2', '3'].map(parseInt) if you want (for explanation, replace parseInt with console.log)).

Use it when either:

When you need to read data written in different radix.
You need to ignore errors (for example change 123px to 123)
Otherwise use other more safe methods (if you need integer, use Math.floor instead).

+number

+ prefix (+number) converts number to float. In case of error it returns NaN which you can compare by either isNaN() or just by number !== number (it should return true only for NaN). It's very fast in Opera.

Use it unless you want specific features of other types.

~~number

~~ is a hack which uses ~ two times on the integer. As ~ bitwise operation can be only done for integers, the number is automatically converted. Most browsers have optimizations for this case. As bitwise operations only work below Math.pow(2, 32) never use this method with big numbers. It's blazingly fast on SpiderMonkey engine.

Use it when either:

You're writing code where performance is important for SpiderMonkey (like FireFox plugins) and you don't need error detection.

You need integer and care resulting JavaScript size.

parseFloat(number)

parseFloat() works like + with the one exception - it processes number until first invalid character instead of returning NaN. It's very fast (but not as fast as ~~ on Firefox) in V8. Unlike parseInt variation, it should be safe with Array.prototype.map.

Use it when either:

You're writing performance-critical code for Node.js or you're writing Google Chrome plugins (V8).

You need to ignore errors (for example change 42.13px to 42.13)

Number(number)

Avoid it. It works just like + prefix and is usually slower. The only usage where it could be useful is callback for Array.prototype.map - you cannot use + as callback.

new Number(number)

Use it when you need to confuse everybody with 0 being truthy value and having typeof of 'number'. Seriously, don't.

Math methods, like Math.ceil(number)

Use them when you need integer as it's more safe than parseInt() by not ignoring unexpected characters. Please note that technically it involves long conversion - string ? float ? integer ? float (numbers in JavaScript are floats) - but most browser have optimizations for it, so usually it's not that noticeable. It's also safe with Array.prototype.map.

Declarations, Names, and Hoisting

In JavaScript, a name enters a scope in one of four basic ways:

1. Language-defined: All scopes are, by default, given the names this and arguments.
2. Formal parameters: Functions can have named formal parameters, which are scoped to the body of that function.
3. Function declarations: These are of the form function foo(){}.
4. Variable declarations: These take the form var foo;.

Function declarations and variable declarations are always moved ("hoisted") invisibly to the top of their containing scope by the JavaScript interpreter. Function parameters and language-defined names are, obviously, already there. This means that code like this:

```
function foo(){
    bar();
    var x = 1;
}
```

is actually interpreted like this:

```
function foo(){
    var x;
    bar();
    x = 1;
}
```

It turns out that it doesn't matter whether the line that contains the declaration would ever be executed. The following two functions are equivalent:

```
function foo(){
    if (false) {
        var x = 1;
    }
    return;
    var y = 1;
}

function foo(){
    var x, y;
    if (false) {
        x = 1;
    }
    return;
```

```
    y = 1;
}
```

Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted. This is not the case with function declarations, where the entire function body will be hoisted as well. But remember that there are two normal ways to declare functions. Consider the following JavaScript:

```
function test(){
    foo(); // TypeError "foo is not a function"
    bar(); // "this will run!"
    var foo = function(){ // function expression assigned to local variable 'foo'
        alert("this won't run!");
    };
    function bar(){ // function declaration, given the name 'bar'
        alert("this will run!");
    }
}
test();
```

In this case, only the function declaration has its body hoisted to the top. The name 'foo' is hoisted, but the body is left behind, to be assigned during execution.

That covers the basics of hoisting, which is not as complex or confusing as it seems. Of course, this being JavaScript, there is a little more complexity in certain special cases.

Name Resolution Order

The most important special case to keep in mind is name resolution order. Remember that there are four ways for names to enter a given scope. The order I listed them above is the order they are resolved in. In general, if a name has already been defined, it is never overridden by another property of the same name. This means that a function declaration takes priority over a variable declaration. This does not mean that an assignment to that name will not work, just that the declaration portion will be ignored. There are a few exceptions:

- The built-in name arguments behaves oddly. It seems to be declared following the formal parameters, but before function declarations. This means that a formal parameter with the name arguments will take precedence over the built-in, even if it is undefined. This is a bad feature. Don't use the name arguments as a formal parameter.
- Trying to use the name this as an identifier anywhere will cause a SyntaxError. This is a good feature.
- If multiple formal parameters have the same name, the one occurring latest in the list will take precedence, even if it is undefined.

Named Function Expressions

You can give names to functions defined in function expressions, with syntax like a function declaration. This does not make it a function declaration, and the name is not brought into scope, nor is the body hoisted. Here's some code to illustrate what I mean:

```
foo(); // TypeError "foo is not a function"
bar(); // valid
baz(); // TypeError "baz is not a function"
spam(); // ReferenceError "spam is not defined"

var foo = function(){}; // anonymous function expression ('foo' gets hoisted)
function bar(){}; // function declaration ('bar' and the function body get hoisted)
var baz = function spam(){}; // named function expression (only 'baz' gets hoisted)

foo(); // valid
bar(); // valid
```

```
baz(); // valid
spam(); // ReferenceError "spam is not defined"
```

How to Code With This Knowledge

Now that you understand scoping and hoisting, what does that mean for coding in JavaScript? The most important thing is to always declare your variables with a var statement. I strongly recommend that you have exactly one var statement per scope, and that it be at the top. If you force yourself to do this, you will never have hoisting-related confusion. However, doing this can make it hard to keep track of which variables have actually been declared in the current scope. I recommend using JSLint with the onevar option to enforce this. If you've done all of this, your code should look something like this:

```
/*jslint onevar: true [...] */
function foo(a, b, c){
    var x = 1,
        bar,
        baz = "something";
}
```

What the Standard Says

I find that it's often useful to just consult the ECMAScript Standard (http://www.ecma-international.org/ecma-262/5.1/) (pdf (http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf)) directly to understand how these things work. Here's what it has to say about variable declarations and scope (section 12.2.2):

If the variable statement occurs inside a FunctionDeclaration, the variables are defined with function-local scope in that function, as described in section 10.1.3. Otherwise, they are defined with global scope (that is, they are created as members of the global object, as described in section 10.1.3) using property attributes { DontDelete }. Variables are created when the execution scope is entered. A Block does not define a new execution scope. Only Program and FunctionDeclaration produce a new scope. Variables are initialized to undefined when created. A variable with an Initializer is assigned the value of its AssignmentExpression when the VariableStatement is executed, not when the variable is created.

Primitive and Reference Types

What are types?

Even though JavaScript has no concept of classes, it still makes use of types. Types are the closest that you will get to classes in JavaScript and there are two kinds: primitive and reference. Primitive types are not stored as objects but rather as simple data types. Reference types are stored as objects, which are really just references to locations in memory rather than a simple value. The tricky part is that JavaScript can make primitive types look like reference types. It does this in order to make the language more consistent for the developer even though it involves a little sleight-of-hand. Primitive values and reference values still behave quite differently even when they appear to be the same.

While other programming languages distinguish between primitive and reference types by storing primitives on the stack and references in the heap, JavaScript does away with this concept completely. Instead, there is the concept of a variable object that tracks variables for a particular scope. Primitive values are stored directly on the variable object while reference values place a pointer in the variable object. That pointer is a reference to a location in memory where the object is stored. There are, of course, other differences between the two types.

Primitive

Primitive types represent simple pieces of data that are stored as-is. There are five primitive types in JavaScript:

- String - a sequence of characters delimited by either single or double quotes. JavaScript has no concept of a character type, so characters are represented as single character strings.
- Number - any numeric value. Both integer and floating-point values are stored using this type.
- Boolean - true or false.
- Null - a primitive type that has only one value, null.
- Undefined - a primitive type has only one value, undefined. This is the value assigned to a variable that is not initialized.

Some examples:

```
// strings
var name = "Andrew";
var selection = "a";

// numbers
var count = 25;
var cost = 1.51;

// boolean
var found = true;

// null
var object = null;

// undefined
var flag = undefined;
var ref; // assigned undefined automatically
```

Primitive values behave in a way that's consistent with many languages. A variable holding a primitive value contains the actual value (as opposed to a pointer to an object). When you assign a primitive value to the variable, the value gets copied into that variable. That means if you assign one variable to another, each variable gets its own copy of the data. For example:

```
var color1 = "red";
var color2 = color1;
```

Here, color1 is assigned the value of "red". The variable color2 is then assigned the value of color1, which creates a new value of "red" and stores it in color2. Now even though color1 and color2 contain the same value, they are completely separate from one another.

```
color1 -> "red"
color2 -> "red"
```

You can change the value in color1 without affecting color2 and vice versa.

Identifying Primitive Types

The best way to identify primitive types is to use the typeof operator. This operator works on any variable and returns a string indicating the type of data. The typeof operator works well with strings, numbers, booleans, and undefined:

```
console.log(typeof "Andrew"); // "string"
console.log(typeof 10); // "number"
console.log(typeof 5.1); // "number"
```

```
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
```

As you might expect, typeof returns "string" when the value is a string, "number" when the value is a number (regardless of integer or floating-point values), "boolean" when the value is a boolean, and "undefined" when the value is undefined. The tricky part involves null.

You wouldn't be the first developer to be confused about the result of this line of code:

```
console.log(typeof null); // "object"
```

When you run typeof null, the result is "object". This has been acknowledged as an error (https://mail.mozilla.org/pipermail/es-discuss/2012-July/024209.html) by TC-39, the committee that designs and maintains JavaScript. You could reason that null is an empty object pointer and therefore "object" is a logical return value, but in reality, this just tends to confuse everybody. The best way to determine if a value is null is to compare it against null directly:

```
console.log(value === null); // true or false
```

Note that this code uses the triple equals operator instead of the double equals operator. Triple equals does the comparison without converting the variable to another type. To understand why this is important, consider the following:

```
console.log("5" == 5); // true
console.log("5" === 5); // false

console.log(undefined == null); // true
console.log(undefined === null); // false
```

Using the double equals, the string "5" and the number 5 are considered equal. That's because the double equals converts the string into a number and then does the comparison. Using the triple equals, however, these values are not considered equal because they are two different types. Likewise, when comparing undefined and null, the double equals says that they are equivalent while the triple equals says they are not. So when trying to identify null, it's important to always use triple equals so that you can correctly identify the type.

Primitive Methods

You may be surprised to learn that strings, numbers, and booleans have methods despite being primitive types (null and undefined do not have any methods). Strings, in particular, have a large number of methods to help you work with them. Some examples:

```
var name = "Andrew";
var lowercaseName = name.toLowerCase(); // convert to lowercase
var firstLetter = name.charAt(0); // get first character
var middleOfName = name.substring(2, 5); // get characters 2-4

var count = 10;
var fixedCount = count.toFixed(2); // convert to "10.00"
var hexCount = count.toString(16); // convert to "a"

var flag = true;
var stringFlag = flag.toString(); // convert to "true"
```

Despite having methods, primitive values themselves are not objects. JavaScript makes it look like these values are objects to provide a consistent experience in the language. Exactly how this works is explained later in this chapter.

Reference Types

Reference types represent objects in JavaScript and are the closest things to classes that you will find in the language. Reference values are instances of reference types and are synonymous with objects. An object is an unordered list of properties consisting of a name and a value. When the value of a property is a function, it is called a method. Because functions themselves are actually reference values in JavaScript, there is little difference between a property that contains a string and a property that contains a function except that a function can be executed. It sometimes helps to think of JavaScript objects as nothing more than hash tables.

```
name -> value
name -> value
```

Objects are said to be instantiated when they are first created, and there are a couple of different ways to create them. The first is to use the new operator with a constructor. A constructor is simply a function that is used to create an object using new. There is no real difference between a constructor and any other function except for how it is used. By convention, constructors in JavaScript begin with a capital letter to distinguish them from non-constructor functions. The following code instantiates a generic object:

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

Reference values behave a bit differently than primitive values. The variable holding a reference value doesn't contain the actual object. Instead, the variable holds a pointer (a.k.a. reference) to a location in memory where the object exists. When you assign a reference value to a variable, you're actually assigning a pointer. That means if you assign one variable to another, each variable gets a copy of the pointer, and both variables still reference the same object in memory. For example:

```
var object1 = new Object();
var object2 = object1;
```

In this code, an object is created and a reference is stored in object1. Then, object2 is assigned the value of object1. There is still only one instance of the object that was created on the first line but both variables now point to the same object.

Two variables pointing to one object

Because JavaScript is a garbage collected language, you don't really need to worry about memory allocations. However, it is best to dereference objects when you no longer need them so the garbage collector can free up that memory. The best way to do this is to set the object variable to null, such as:

```
var object1 = new Object();
// do something
object1 = null; // dereference
```

In this example, object1 is created and used before finally being set to null. When there are no more references to an object in memory, the garbage collector is able to free up that memory. Dereferencing objects becomes important in very large applications that use millions of objects.

Another interesting aspect of objects in JavaScript is that you can add and remove properties at any time. For example:

```
var object1 = new Object();
var object2 = object1;

object1.myCustomProperty = "Awesome!";
console.log(object2.myCustomProperty)); // "Awesome!"
```

Here, myCustomProperty is added to object1 with a value of "Awesome!". Because both object1 and object2 point to the same object, that property is also accessible on object2. This is one of the unique aspects of JavaScript: it allows you to modify objects whenever you want even if you didn't define the objects in the first place. There are

ways to prevent such modification and those are discussed later in this book.

Built-In Types

There are several built-in reference types in JavaScript that can be instantiated at any time:

- Array - an ordered list of numerically indexed values.
- Date - a date and time.
- Error - a runtime error. There are also several subtypes of this type for more specific errors.
- Function - a function.
- Object - a generic object.
- RegExp - a regular expression.

Each of these built-in reference types can be instantiated using new, such as:

```
var items = new Array();
var now = new Date();
var error = new Error("Something bad happened.");
var func = new Function("console.log('Hi');");
var object = new Object();
var re = new RegExp("\\d+");
```

Several of these reference types also have literal forms. A literal is special syntax that allows you to define a value, both primitive and reference, without explicitly creating an object. Earlier in this chapter, you saw string literals, number literals, boolean literals, the null literal, and the undefined literal. Those are all primitive literals. Reference literals allow you to define objects without using their constructor.

For example, you can create an object (similar to new Object()) by using object literal syntax. An object literal is made up of curly braces with properties defined inside. Properties are made up of an identifier or string, a colon, and a value. Multiple properties are separated by commas. For example:

```
// "Awesome!"

var book = {
    name: "Principles of Object-Oriented Programming in JavaScript",
    year: 2012
};
```

You can also use string literals as property names, Which allow you to have property names with spaces or other special characters:

```
var book = {
    "name": "Principles of Object-Oriented Programming in JavaScript",
    "year": 2012
};
```

Both of these examples are logically equivalent to the following:

```
var book = new Object();
book.name = "Principles of Object-Oriented Programming in JavaScript";
book.year = 2012;
```

Keep in mind that using an object literal doesn't actually call new Object(). Instead, the JavaScript engine follows the same steps without actually using the constructor. This is true for all reference literals.

Similarly, you can define an array literal by enclosing any number of values, separated by commas, inside of square brackets. For example:

```
var colors = ["red", "blue", "green"];
console.log(colors[0]); // "red"
```

```
// equivalent

var colors = new Array("red", "blue", "green");
console.log(colors[0]); // "red"
```

Functions, of course, are almost always defined using their literal form. In fact, it's discouraged to use the Function constructor at all. It's much easier and far less error-prone to use the literal form.

For example:

```
function reflect(value) {
    return value;
}

// same as

var reflect = new Function("value", "return value;");
```

This code defines the function called reflect() that simply passes back the value that is passed in. Even for this simple function, the literal form is much easier to write and understand than the constructor form. Further, there is no good way to debug functions that are created in the constructor form. You'll only ever rarely see code using the Function constructor.

JavaScript also has regular expression literals that allow you to define regular expressions without using the RegExp constructor. Regular expression literals look very similar to regular expressions in Perl. The pattern itself is contained between two slashes and any additional options are single characters after the second slash. For example:

```
var numbers = /\d+/g;
// same as
var numbers = new RegExp("\\d+", "g");
```

The literal form of regular expressions in JavaScript is a little bit easier to deal with because you don't need to worry about escaping characters within strings. When using the RegExp constructor, the pattern is passed in as a string and therefore backslashes must be escaped (which is why \d is used in the literal and \\d is used in the string). Regular expression literals are used most of the time in JavaScript except in cases where the regular expression is being constructed dynamically from one or more strings.

With the exception of Function, there really isn't any right or wrong way to instantiate built-in types. Many developers prefer using literals while some prefer using constructors. The choice of which to use is completely up to you.

Property Access

As discussed previously, properties are name-value pairs stored on an object. The primary way that properties are accessed on objects is through dot notation. Dot notation is very common in object-oriented languages and is the most frequently used property access pattern in JavaScript. However, it's also possible to access properties on JavaScript objects using bracket notation with a string. Consider this code using dot notation:

```
var array = [];
array.push(12345);

// bracket notation

var array = [];
array["push"](12345);
```

Note that the name of the method is now included in a string that's enclosed in square brackets. This syntax may seem a little strange, but is very useful when you want to dynamically decide what property to access. For example:

```
var array = [];
var method = "push";
array[method](12345);
```

Bracket notation allows you to use a variable instead of the string literal to specify the property to access. Here, the variable method has a value of "push", so push() ends up being called on the array. This capability comes is quite useful and will be used several times throughout this book. The important thing to keep in mind is that there is no difference, performance or otherwise, between using dot notation and bracket notation outside of the syntax. Developers tend to find dot notation easier to read and so it is used more frequently than bracket notation.

Identifying Reference Types

The easiest reference type to identify is a function because you can use the typeof operator. When used with a function, typeof returns "function":

```
function reflect(value) {
    return value;
}

console.log(typeof reflect); // "function"
```

Other reference types are a little bit trickier because typeof doesn't provide any additional detail. For all reference types except functions, typeof returns "object". That's not very helpful when you're dealing with a lot of different types.

To make identifying reference types easier, JavaScript has the instanceof operator. The instanceof operator is used with a value and a constructor. When the value is an instance of the type identified with the constructor, then instanceof returns true, otherwise it returns false. Here are some examples:

```
var items = [];
var object = {};

function reflect(value) {
    return value;
}

console.log(items instanceof Array); // true
console.log(object instanceof Object); // true
console.log(object instanceof Array); // false
console.log(reflect instanceof Function); // true
```

It's also important to understand that instanceof works with inheritance. That means every object is actually an instance of Object because every reference type inherits from Object:

```
var items = [];
var object = {};

function reflect(value) {
return value;
}

console.log(items instanceof Array); // true
console.log(items instanceof Object); // true
console.log(object instanceof Object); // true
console.log(object instanceof Array); // false
console.log(reflect instanceof Function); // true
console.log(reflect instanceof Object); // true
```

Inheritance is discussed later in the book, but for now, all you need to know is that all objects inherit from Object.

Although instanceof can be used to identify arrays, there is one exception that affects web developers specifically. JavaScript values can be passed back and forth between frames in the same webpage. That's normally not a problem unless you try to identify the type of a reference value. Since each webpage has its own global context, that means each webpage has its own version of Object, Array, and all other built-in types. So if you pass an array from one frame into another frame, using instanceof doesn't work because the array is actually an instance of Array from a different frame.

To solve this problem, ECMAScript 5 introduced Array.isArray(), which definitively identifies the value as an instance of Array regardless of the value's origin. This method returns true when a value is passed in that is a native array from any context. If you are using ECMAScript 5 compliant environment, then Array.isArray() is the best way to identify arrays:

```
var items = [];
console.log(Array.isArray(items)); // true
```

The Array.isArray() method is supported in Node.js, WebKit-based browsers, Gecko-based browsers, Opera, and Internet Explorer 9+.

Primitive Wrapper Types

Perhaps the most confusing part of JavaScript is the concept of primitive wrapper types. There are three primitive wrapper types: String, Number, and Boolean. These primitive wrapper types exist because JavaScript aims to make working with primitive values as easy as working with objects. It would be very confusing if you had to use a different syntax or switch to a procedural style just to get a substring of text.

The primitive wrapper types are reference types just like any other, however, they are automatically created behind-the-scenes whenever strings, numbers, or booleans are read. Consider the following example:

```
var name = "Andrew";
var firstChar = name.charAt(0);
console.log(firstChar); // "A"
```

On the first line of code, a primitive string value is assigned to name. In the second line, name is used as if it were an object and charAt(0) is called using dot notation. Behind-the-scenes, this is what actually is happening:

```
var name = "Andrew";
var temp = new String(name);
var firstChar = temp.charAt(0);
temp = null;
console.log(firstChar); // A
```

Because the second line is using a string as if it's an object, the JavaScript engine creates an instance of String so that charAt(0) will work. The String object exists only for a single statement before it is destroyed, a process called autoboxing. You can test this out by trying to add a property to a string as if it were a regular object:

```
var name = "Andrew";
name.last = "Gerst";
console.log(name.last); // undefined
```

This code attempts to add a property called last to the string name. There is nothing wrong with this code, either syntactically or otherwise, except that the property disappears. With regular objects, you can add properties at any time and they stay until you manually remove them. With primitive wrapper types, properties seem to disappear because the object on which the property was assigned is actually destroyed. Here's what's actually happening in the JavaScript engine:

```
// What the JavaScript engine does
var name = "Andrew";
var temp = new String(name);
temp.last = "Gerst";
temp = null;
console.log(name.last); // undefined
```

So instead of assigning a new property to a string, the code actually creates a new property on a temporary object that is then destroyed.

Since reference values are created automatically for primitive values, that means you can actually use instanceof to check for these types of values;

```
var name = "Andrew";
var count = 10;
var found = false;

console.log(name instanceof String); // true
console.log(count instanceof Number); // true
console.log(found instanceof Boolean); // true
```

The reason this works is because of the temporary object being created with the primitive value is read. The temporary object is, technically, an instance of the particular primitive wrapper type. It is still best to use typeof when trying to identify primitive values, but it is important to understand why instanceof works in this way.

Because primitive wrapper types are reference types just like any other, you can actually create them manually as well. However, there are some side effects:

```
var name = new String("Andrew");
var count = new Number(10);
var found = new Boolean(false);

console.log(typeof name); // "object"
console.log(typeof count); // "object"
console.log(typeof found); // "object"
```

When you create an instance of the primitive wrapper type, you have now created an object just like any other. That means typeof can no longer identify the type of data you are intending to store. Also, you cannot use these objects the same way that you use primitive values. The best example is with a Boolean object:

```
var found = new Boolean(false);

if (found) {
    console.log("Found"); // this executes
}
```

There are other confusing circumstances when you instantiate primitive wrappers on your own. It's recommended to avoid doing so unless you have figured out the special case where it makes sense. Most of the time, using primitive wrapper objects instead of primitives will only lead to errors.

Summary

While JavaScript has no formal concept of classes, it does have the concept of types. Each variable or piece of data has a specific type associated with it. That type can be a primitive or a reference. There are five primitive types: strings, numbers, booleans, null, and undefined. These represent simple values that are stored directly in the variable object for a given context. Primitive types can be identified using typeof with the exception of null, which must be compared directly against the special value null.

Reference types are the closest thing to classes in JavaScript. Objects are said to be instances of reference

types. You can create new objects by using the new operator or by using a reference literal. Properties and methods are primarily accessed using dot notation but can also be accessed using bracket notation. Functions are objects in JavaScript just like anything else, it can be identified by using the typeof operator. Other reference types should use instanceof with a constructor to determine the type of object.

In order to make primitives seem more like references, JavaScript has three primitive wrapper types: String, Number, and Boolean. These are used whenever you access a primitive value corresponding to one of these types. These objects are created behind-the-scenes so that you can access methods on primitives as if they were regular objects. However, those objects are only temporary and are destroyed as soon as statement is complete. While it's possible to create your own instances of primitive wrappers, it's not recommended because it can lead to confusing situations.

Chaining Methods

In the world of OOP, the previous ways of defining an object is too limiting in many situations. We need a way to create an object "type" that can be used multiple times without having to redefine the object every time to meet each particular instance's needs. The standard way to achieve this is to use the Object Constructor function.

An object constructor is merely a regular JavaScript function, so it's just as robust (ie: define parameters, call other functions etc). The difference between the two is that a constructor function is called via the new operator (which you'll see below). By basing our object definition on the function syntax, we get its robustness as well.

Lets use a real world item "cat" as an example. A property of a cat may be its color or name. A method may be to "meeyow". The important thing to realize, however is that every cat will have a different name or even meeyow noise. To create an object type that accommodates this need for flexibility, we'll use an object constructor.

Here the function "Cat" is an object constructor, and its properties and methods are declared inside it by prefixing them with the keyword "this." Objects defined using an object constructor are then instantiated using the new keyword. Notice how we're able to easily define multiple instances of cat, each with its own name- that's the flexibility object constructor brings to custom objects. Constructors create the blueprints for objects, not the object itself.

Object Constructors are capitalized by convention. A constructor name should begin with an uppercase letter.

```
function Cat(name){
    this.name = name;
    this.talk = function(){
        log(this.name + " say meeow!");
    };
}

cat1 = new Cat("felix");
cat1.talk(); // "felix says meeow!"

cat2 = new Cat("ginger");
cat2.talk(); // "ginger says meeow!"

Cat.prototype.changeName = function(name){
    this.name = name;
};

firstCat = new Cat("pursur");
firstCat.changeName("bill");
firstCat.talk(); // "bill says meeow!"

// What we want to do (chaining) methods:
new Cat("kitty").changeName("hal buttar").talk();
```

```
// Uncaught TypeError: Cannot call method 'talk' of undefined.
// Remember, all functions return "undefined" if you don't specify what should be returned.
```

```
cat1 -> Array
(
    [name] => felix
    [talk] => function (){
                log(this.name + " say meeow!");
        }
    [changeName] => function (name){
        this.name = name;
}
)

cat2 -> Array
(
    [name] => ginger
    [talk] => function (){
                log(this.name + " say meeow!");
        }
    [changeName] => function (name){
        this.name = name;
}
)

firstCat -> Array
(
    [name] => bill
    [talk] => function (){
                log(this.name + " say meeow!");
        }
    [changeName] => function (name){
        this.name = name;
}
)

log -> Array
(
    [0] => felix say meeow!
    [1] => ginger say meeow!
    [2] => bill say meeow!
)
```

We saw above how to add a method to our constructor function by merely declaring it inside the function. Another approach is through prototyping, which is also more popular due to its elegance. Prototype is a type of inheritance in JavaScript. We use it when we would like an object to inherit a method after it has been defined. Think of prototyping mentally as "attaching" a method to an object after it's been defined, in which all object instances then instantly share.

As you can see we merely use the keyword "prototype" immediately following the object's name to utilize this functionality. The custom method changeName() is now shared by all instances of cat.

Prototyping works on both custom objects and select prebuilt objects, such as Date() or String. For the later, the general rule is that you can prototype any prebuilt object that's initialized with the "new" keyword.

Right now we can't chain the methods together but we can easily fix this by returning this inside the cat object's methods.

```
function Cat(name){
```

```
    this.name = name;
    this.talk = function(){
        log(this.name + " say meeow!");
        return this;
    };
}

Cat.prototype.changeName = function(name){
    this.name = name;
    return this;
};

cat1 = new Cat("felix");
cat1.talk().talk().changeName("newFelix").talk();

cat2 = new Cat("ginger");
cat2.talk().talk().changeName("newGinger").talk();

firstCat = new Cat("pursur");
firstCat.changeName("newPursur").talk();

new Cat("hal buttar").talk().changeName("new hal buttar").talk().talk();
new Cat().talk().changeName("anonymous").talk();
```

```
cat1 -> Array
(
    [name] => newFelix
    [talk] => function (){
                log(this.name + " say meeow!");
                return this;
        }
    [changeName] => function (name){
        this.name = name;
        return this;
}
)

cat2 -> Array
(
    [name] => newGinger
    [talk] => function (){
                log(this.name + " say meeow!");
                return this;
        }
    [changeName] => function (name){
        this.name = name;
        return this;
}
)

firstCat -> Array
(
    [name] => newPursur
    [talk] => function (){
                log(this.name + " say meeow!");
                return this;
        }
    [changeName] => function (name){
        this.name = name;
        return this;
```

```
}
)

log -> Array
(
    [0] => felix say meeow!
    [1] => felix say meeow!
    [2] => newFelix say meeow!
    [3] => ginger say meeow!
    [4] => ginger say meeow!
    [5] => newGinger say meeow!
    [6] => newPursur say meeow!
    [7] => hal buttar say meeow!
    [8] => new hal buttar say meeow!
    [9] => new hal buttar say meeow!
    [10] => undefined say meeow!
    [11] => anonymous say meeow!
)
```

Saving State

You can save state by using window.location.hash or window.location.search as they both are in the url.

Changing location.search forces a page refresh which is most likely undesirable. Updating location.hash doesn't reload the page.

Creating shortcuts

bind() is helpful in cases where you want to create a shortcut to a function which requires a specific this value.

Take Array.prototype.slice, for example, which you want to use for converting an array-like object to a real array. You could create a shortcut like this:

```
var slice = Array.prototype.slice;
slice.call(arguments);
```

With bind(), this can be simplified. In the following piece of code, slice is a bound function to the call() (https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/call) function of Function.prototype, with the this value set to the slice() function of Array.prototype. This means that additional call() calls can be eliminated:

```
var unboundSlice = Array.prototype.slice; // same as "slice" in the previous example
var slice = Function.prototype.call.bind(unboundSlice);

slice(arguments);
```

Statements and Expressions

All the JavaScript code that you will write will, for the most part, be comprised of many separate statements. A statement can set a variable equal to a value. A statement can also be a function call, i.e. document.write(). Statements define what the script will do and how it will be done.

In typical programming languages like C and PHP, the end of a statement is marked with a semicolon(;), but we have seen that the semicolon is optional in JavaScript. In JavaScript, the end of a statement is most often marked by pressing return and starting a new line. If you are an experienced programmer and prefer to use semicolons, feel free to do so. JavaScript will not malfunction from ending semicolons. The only time it is necessary to use a semicolon is when you choose to smash two statements onto one line (i.e. two document.write statements on one line).

In addition to standard statements like changing a variable's value, assigning a new value, or calling a function, there are groups of statements that are distinct in their purpose. These distinct groups of statements include:

- Conditional Statements
- Loop Statements
- Object Manipulation Statements
- Comment Statements
- Exception Handling Statements

The function Statement Versus the function Expression

JavaScript has a function statement as well as a function expression. This is confusing because they can look exactly the same. A function statement is shorthand for a var statement with a function value.

The statement:

```
function foo(){}
// means about the same thing as:
var foo = function foo(){};
```

The second form makes it clear that foo is a variable containing a function value. To use the language well, it is important to understand that functions are values.

function statements are subject to hoisting. This means that regardless of where a function is placed, it is moved to the top of the scope in which it is defined. This relaxes the requirement that functions should be declared before used, which I think leads to sloppiness. It also prohibits the use of function statements in if statements. It turns out that most browsers allow function statements in if statements, but they vary in how that should be interpreted. That creates portability problems.

The first thing in a statement cannot be a function expression because the official grammar assumes that a statement that starts with the word function is a function statement. The workaround is to wrap the function expression in parentheses:

```
(function(){
    var hidden_variable;
    // This function can have some impact on the environment, but introduces no new global variables.
})();
```

ECMAScript 5 Strict Mode

Strict Mode is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a "strict" operating context. This strict context prevents certain actions from being taken and throws more exceptions (generally providing the user with more information and a tapered-down coding experience).

Since ECMAScript 5 is backwards-compatible with ECMAScript 3, all of the "features" that were in ECMAScript 3 that were "deprecated" are just disabled (or throw errors) in strict mode, instead.

Strict mode helps out in a couple ways:

- It catches some common coding bloopers, throwing exceptions.

- It prevents, or throws errors, when relatively "unsafe" actions are taken (such as gaining access to the global object).
- It disables features that are confusing or poorly thought out.

It should be noted that ECMAScript 5's strict mode is different from the strict mode available in Firefox (which can be turned on by going to about:config and enabled javascript.options.strict). ES5's strict mode complains about a completely different set of potential errors (whereas Firefox's existing strict mode tries to enforce some good practices, only).

How do you enable strict mode?

```
// Simple. Toss this at the top of a program to enable it for the whole script:
"use strict";

// Or place it within a function to turn on strict mode only within that context.
function imStrict(){
    "use strict";
    // ... your code ...
}
```

Note the syntax that's used to enable strict mode (I love this!). It's simply a string in a single statement that happens to contain the contents "use strict". *No new syntax is introduced in order to enable strict mode.* This is huge. This means that you can turn strict mode on in your scripts - today - and it'll have, at worst, no side effect in old browsers.

As you may note from the examples here and in the previous post there are virtually no new syntax additions or changes to the language in ECMAScript 5. This means that you can write your ES5 scripts in a manner that will be able to gracefully degrade for older useragents - something that wasn't possible with ECMAScript 4. The way in which strict mode is enabled is a great illustration of that point in practice.

A neat aspect of being able to define strict mode within a function is that you can now define complete JavaScript libraries in a strict manner without affecting outside code.

```
// Non-strict code...

(function(){
    "use strict";

    // Define your library strictly...
})();

// Non-strict code...
```

A number of libraries already use the above technique (wrapping the whole library with an anonymous self-executing function) and they will be able to take advantage of strict mode very easily.

So what changes when you put a script into strict mode? A number of things.

Variables and Properties

An attempt to assign *foo = "bar";* where 'foo' hasn't been defined will fail. Previously it would assign the value to the foo property of the global object (e.g. *window.foo*), now it just throws an exception. This is definitely going to catch some annoying bugs.

Any attempts to write to a property whose writable attribute is set to false, delete a property whose configurable attribute is set to false, or add a property to an object whose extensible attribute is set to false will result in an error (these attributes were discussed previously). Traditionally no error will be thrown when any of these

actions are attempted, it will just fail silently.

Deleting a variable, a function, or an argument will result in an error.

```
var foo = "test";
function test(){}
delete foo; // Error
delete test; // Error

function test2(arg){
    delete arg; // Error
}
```

Defining a property more than once in an object literal will cause an exception to be thrown.

```
// Error
{ foo: true, foo: false }
```

eval

Virtually any attempt to use the name 'eval' is prohibited - as is the ability to assign the eval function to a variable or a property of an object.

```
// All generate errors...
obj.eval = ...
obj.foo = eval;
var eval = ...;
for (var eval in ...) {}
function eval(){}
function test(eval){}
function(eval){}
new Function("eval")
```

Additionally, attempts to introduce new variables through an eval will be blocked.

```
eval("var a = false;");
print(typeof a); // undefined
```

Functions

Attempting to overwrite the arguments object will result in an error:
arguments = [...]; // not allowed

Defining identically-named arguments will result in an error *function(foo, foo){}*.

Access to *arguments.caller* and *arguments.callee* now throw an exception. Thus any anonymous functions that you want to reference will need to be named, like so:

```
setTimeout(function later(){
    // do stuff...
    setTimeout(later, 1000);
}, 1000);
```

The *arguments* and *caller* properties of other functions no longer exist - and the ability to define them is prohibited.

```
function test(){
    function inner(){
        // Don't exist, either
        test.arguments = ...; // Error
        inner.caller = ...; // Error
    }
}
```

Finally, a long-standing (and very annoying) bug has been resolved: Cases where null or undefined is coerced into becoming the global object. Strict mode now prevents this from happening and throws an exception instead.

```
(function(){ ... }).call(null); // Exception
```

with () {}

with () {} statements are dead when strict mode is enabled - in fact it even appears as a syntax error. While the feature was certainly mis-understood and possibly mis-used I'm not convinced that it's enough to be stricken from the record.

The changes made in ECMAScript 5 strict mode are certainly varied (ranging from imposing stylistic preferences, like removing with statements, to fixing legitimately bad language bugs, like the ability to redefine properties in object literals). It'll be interesting to see how people begin to adopt these points and how it'll change JavaScript development.

ECMAScript 6 (Harmony)

Version 6 is rumored to have support for classes, a concept long supported by languages like Java, C++ and C#, in addition to multiple new concepts and language features.

In the July 2008 announcement, Eich also stated that the ECMAScript 4 proposal would be superseded by a new project, code-named ECMAScript Harmony. ECMAScript Harmony (http://people.mozilla.org/~jorendorff/es6-draft.html) names the agreed design trajectory of post-ES5 editions. It will include syntactic extensions, but the changes will be more modest than ECMAScript 4 in both semantic and syntactic innovation. Packages, namespaces, and early binding from ECMAScript 4 are no longer included for planned releases. In addition, other goals and ideas from ECMAScript 4 are being rephrased to keep consensus in the committee; these include a notion of classes based on ECMAScript, 5th Edition (being an update to ECMAScript, 3rd edition). As of December 2009, there is no publicly announced release date for next edition within the ECMAScript Harmony trajectory. Depending on ECMA, that next edition may end up being called ECMAScript, 6th edition.

Object.create(proto [, propertiesObject])

Objects can also be created using the Object.create method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function.

```
// create an object with null as prototype
o = Object.create(null);
```

```
o = {};
// is equivalent to
o = Object.create(Object.prototype);

// Example where we create an object with a couple of sample properties.
// (Note that the second parameter maps keys to *property descriptors*.)
o = Object.create(Object.prototype, {
    // foo is a regular "value property"
    foo: { writable: true, configurable: true, value: "hello" },
    // bar is a getter-and-setter (accessor) property
    bar: {
        configurable: false,
        get: function(){ return 10 },
        set: function(value){ console.log("Setting `o.bar` to", value) }
    }
});

function Constructor(){}
o = new Constructor();
// is equivalent to
o = Object.create(Constructor.prototype);
// Of course, if there is actual initialization code in the Constructor function, the Object.create cannot reflect it

// create a new object whose prototype is a new, empty object
// and a adding single property 'p', with value 42
o = Object.create({}, { p: { value: 42 } });

// by default properties ARE NOT writable, enumerable, or configurable
o.p = 24;
o.p // 42

o.q = 12;
for (var prop in o) {}
    console.log(prop)
}
// "q"

delete o.p // false

// to specify an ES3 property
o2 = Object.create({}, { p: { value: 42, writable: true, enumerable: true, configurable: true } });
```

There are 2 kinds of properties: Data properties and Accessor properties. The difference between the 2 is that accessor ones have get and set attributes. Looking into a property, we can define it as a named collection of attributes. Every property has a value which is the value that you assign from it, that you get back from it, and it has three Boolean flags: writable, enumerable, and configurable, which control whether the thing is read-only or if it's enumerated by for in, or if you can delete it or change it. These flags are there in the language but never exposed to the programmer. This was fixed in ES5 and this is how you can use them:

```
Object.defineProperty(object, key, descriptor)
Object.defineProperties(object, object_of_descriptors)
Object.getOwnPropertyDescriptor(object, key)
Object.getOwnProperties(object)
Object.keys(object)
```

A usage example: In ES3, we use object literals that looks like:

```
var obj = { foo: bar };
```

Notice that creating an object this way, strips the developer from any control over the created object. This has changed in ES5, where we can set those control flags mentioned earlier by doing:

```
var myObj = Object.defineProperties(Object.create(Object.prototype), {
    foo: {
        value: bar,
        writable: true,
        enumerable: true,
        configurable: true
    }
});
```

ES5 has also provided: Accessor properties get and set that can be used like:

```
Object.defineProperty(obj, 'name', {
    get: function(){ return this.firstName + " " + this.lastName },
    set: function(value){ this.firstName = value.split(' ')[0]; this.lastName = value.split(' ')[1] },
    enumerable: true
});
```

Bitwise Operators

This is a little bit of a tangent; however, I think it's important not to get thrown off when we see code that uses bitwise operators. Bitwise operators are good for saving space -- but many times, space is hardly an issue.

Another example comes up when dealing with data compression: what if you wanted to compress a file? In principle, this means taking one representation and turning it into a representation that takes less space. One way of doing this is to use an encoding that takes less than 8 bits to store a byte. (For instance, if you knew that you would only be using the 26 letters of the Roman alphabet and didn't care about capitalization, you'd only need 5 bits to do it.) In order to encode and decode files compressed in this manner, you need to actually extract data at the bit level. Finally, you can use bit operations to speed up your program or perform neat tricks. (This isn't always the best thing to do.)

Thinking about Bits

The byte is the lowest level at which we can access data; there's no "bit" type, and we can't ask for an individual bit. In fact, we can't even perform operations on a single bit -- every bitwise operator will be applied to, at a minimum, an entire byte at a time. This means we'll be considering the whole representation of a number whenever we talk about applying a bitwise operator. (Note that this doesn't mean we can't ever change only one bit at a time; it just means we have to be smart about how we do it.) Understanding what it means to apply a bitwise operator to an entire string of bits is probably easiest to see with the shifting operators. By convention, in C and C++ you can think about binary numbers as starting with the most significant bit to the left (i.e., 10000000 is 128, and 00000001 is 1). Regardless of underlying representation, you may treat this as true. As a consequence, the results of the left and right shift operators are not implementation dependent for unsigned numbers (for signed numbers, the right shift operator is implementation defined).

The leftshift operator is the equivalent of moving all the bits of a number a specified number of places to the left:

[variable]<<[number of places]

For instance, consider the number 8 written in binary 00001000. If we wanted to shift it to the left 2 places, we'd end up with 00100000; everything is moved to the left two places, and zeros are added as padding. This is the number 32 -- in fact, left shifting is the equivalent of multiplying by a power of two.

```
function mult_by_pow_2(number, power){
    return number<<power;
}
```

Note that in this example, we're using integers, which are either 2 or 4 bytes, and that the operation gets applied to the entire sequence of 16 or 32 bits.

But what happens if we shift a number like 128 and we're only storing it in a single byte: 10000000? Well, 128 * 2 = 256, and we can't even store a number that big in a byte, so it shouldn't be surprising that the result is 00000000.

It shouldn't surprise you that there's a corresponding right-shift operator: >> (especially considering that I mentioned it earlier). Note that a bitwise right-shift will be the equivalent of integer division by 2.

Why is it integer division? Consider the number 5, in binary, 00000101. 5/2 is 2.5, but if you are performing integer division, 5/2 is 2. When you perform a right shift by one: (unsigned int) 5 >> 1, you end up with 00000010, as the rightmost 1 gets shifted off the end; this is the representation of the number 2. Note that this only holds true for unsigned integers; otherwise, we are not guaranteed that the padding bits will be all 0s.

Generally, using the left and right shift operators will result in significantly faster code than calculating and then multiplying by a power of two. The shift operators are also be useful for manipulating individual bits.

A bitwise operation operates on one or more bit patterns or binary numerals at the level of their individual bits. It is a fast, primitive action directly supported by the processor, and is used to manipulate values for comparisons and calculations. On simple low-cost processors, typically, bitwise operations are substantially faster than division, several times faster than multiplication, and sometimes significantly faster than addition. While modern processors usually perform addition and multiplication just as fast as bitwise operations due to their longer instruction pipelines and other architectural design choices, bitwise operations do commonly use less power/performance because of the reduced use of resources.

The bitwise operators operate on numbers (always integers) as if they were sequences of binary bits (which, of course, internally to the computer they are). These operators will make the most sense, therefore, if we consider integers as represented in binary, octal, or hexadecimal (bases 2, 8, or 16), not decimal (base 10). Remember, you can use octal constants by prefixing them with an extra 0 (zero), and you can use hexadecimal constants by prefixing them with 0x (or 0X).

The & operator performs a bitwise AND on two integers. Each bit in the result is 1 only if both corresponding bits in the two input operands are 1. For example, 0x56 & 0x32 is 0x12, because (in binary):

```
  0 1 0 1 0 1 1 0
& 0 0 1 1 0 0 1 0
  ---------------
  0 0 0 1 0 0 1 0
```

The | (vertical bar / pipe) operator performs a bitwise OR on two integers. Each bit in the result is 1 if either of the corresponding bits in the two input operands is 1. For example, 0x56 | 0x32 is 0x76, because:

```
  0 1 0 1 0 1 1 0
| 0 0 1 1 0 0 1 0
  ---------------
  0 1 1 1 0 1 1 0
```

The ^ (caret) operator performs a bitwise exclusive-OR on two integers. Each bit in the result is 1 if one, but not both, of the corresponding bits in the two input operands is 1. For example, 0x56 ^ 0x32 is 0x64:

```
  0 1 0 1 0 1 1 0
^ 0 0 1 1 0 0 1 0
  --------------
  0 1 1 0 0 1 0 0
```

The ~ (tilde) operator performs a bitwise complement on its single integer operand. (The ~ operator is therefore a unary operator, like ! and the unary -, &, and * operators.) Complementing a number means to change all the 0 bits to 1 and all the 1s to 0s. For example, assuming 16-bit integers, ~0x56 is 0xffa9:

```
~ 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0
  ------------------------------
  1 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1
```

The << operator shifts its first operand left by a number of bits given by its second operand, filling in new 0 bits at the right. Similarly, the >> operator shifts its first operand right. If the first operand is unsigned, >> fills in 0 bits from the left, but if the first operand is signed, >> might fill in 1 bits if the high-order bit was already 1. (Uncertainty like this is one reason why it's usually a good idea to use all unsigned operands when working with the bitwise operators.) For example, 0x56 << 2 is 0x158:

```
        0 1 0 1 0 1 1 0 << 2
  ------------------
  0 1 0 1 0 1 1 0 0 0
```

And 0x56 >> 1 is 0x2b:

```
  0 1 0 1 0 1 1 0 >> 1
  --------------
       0 1 0 1 0 1 1
```

For both of the shift operators, bits that scroll "off the end'' are discarded; they don't wrap around. (Therefore, 0x56 >> 3 is 0x0a.)

The bitwise operators will make more sense if we take a look at some of the ways they're typically used. We can use & to test if a certain bit is 1 or not. For example, 0x56 & 0x40 is 0x40, but 0x32 & 0x40 is 0x00:

```
  0 1 0 1 0 1 1 0        0 0 1 1 0 0 1 0
& 0 1 0 0 0 0 0 0      & 0 1 0 0 0 0 0 0
  --------------         --------------
  0 1 0 0 0 0 0 0        0 0 0 0 0 0 0 0
```

Because of the nature of base-2 arithmetic, it turns out that shifting left and shifting right are equivalent to multiplying and dividing by two. These operations are equivalent for the same reason that tacking zeroes on to the right of a number in base 10 is the same as multiplying by 10, and deleting digits from the right is the same as dividing by 10. You can convince yourself that 0x56 << 2 is the same as 0x56 * 4, and that 0x56 >> 1 is the same as 0x56 / 2. It's also the case that masking off all but the low-order bits is the same as taking a remainder; for example, 0x56 & 0x07 is the same as 0x56 % 8. Some programmers therefore use <<, >>, and & in preference to *, /, and % when powers of two are involved, on the grounds that the bitwise operators are "more efficient." Usually it isn't worth worrying about this, though, because most compilers are smart enough to perform these optimizations anyway (that is, if you write x * 4, the compiler might generate a left shift instruction all by itself), they're not always as readable, and they're not always correct for negative numbers.

The issue of negative numbers, by the way, explains why the right-shift operator >> is not precisely defined when the high-order bit of the value being shifted is 1. For signed values, if the high-order bit is a 1, the number is negative. (This is true for 1's complement, 2's complement, and sign-magnitude representations.) If you were using a right shift to implement division, you'd want a negative number to stay negative, so on some computers, under some compilers, when you shift a signed value right and the high-order bit is 1, new 1 bits are shifted in at the left instead of 0s. However, you can't depend on this, because not all computers and compilers implement right shift this way. In any case, shifting negative numbers to the right (even if the high-order 1 bit propagates) gives you an incorrect answer if there's a remainder involved: in 2's complement, 16-bit arithmetic, -15 is 0xfff1, so -15 >> 1 might give you 0xfff8 shifted which is -8. But integer division is supposed to discard the remainder, so -15 / 2 would have given you -7. (If you're having trouble seeing the way the shift worked, 0xfff1 is 1111111111110001_2 and 0xfff8 is 1111111111111000_2. The low-order 1 bit got shifted off to the right, but because

the high-order bit was 1, a 1 got shifted in at the left.)

You know a couple of neat tricks that you can use when performance is critical, or space is slow, or you just need to isolate and manipulate individual bits of a number. You now should have a better sense of what goes on at the lowest levels of your computer.

One final neat trick of bitwise operators is that you can use them, in conjunction with a bit of math, to find out whether an integer is a power of two.

A power of two will look like this in memory:

01000000

a string of zeros, with a lone one. Now, if you subtract 1 from a power of two, you'll get, with all numbers in binary:

01000000 - 00000001 = 00111111

a string of ones!

If you take the bitwise AND of the two values, you get 0:

01000000 & 00111111 = 00000000

On the other hand, if you don't have a power of two, you'll have at least one additional 1:

01000001

When you subtract 1, you'll still have at least one "on" bit (with a value of 1) in the same position as before, so taking the bitwise AND of the two numbers will not result in a string of 0s:

01000001 - 00000001 = 01000000
01000000 & 01000001 = 01000000

So to tell if an integer is a power of two:

```
function is_power(x){
    return !((x-1) & x);
}
```

Math.round ~~ hack

Math.round() has a function call overhead, so using the ~~ hack (truncate towards 0) and adding 0.5 works quicker, but if you want to handle negative numbers too then you have to check whether to add or subtract 0.5 .... and this wipes out the speed advantage. So ~~(0.5+num) is only worth it if you know your numbers always have the same sign...

Here is the jsperf test of these methods.
http://jsperf.com/math-round-vs-hack (http://jsperf.com/math-round-vs-hack)
http://jsperf.com/bit-vs-comparison-operator (http://jsperf.com/bit-vs-comparison-operator)

```
var somenum = -500 + (Math.random() * 1000), rounded;

// "proper" rounding
rounded = Math.round(somenum);

// Hack rounding
```

```
rounded = ~~ (0.5 + somenum);

// Proper hack rounding
rounded = ~~ (somenum + (somenum > 0 ? .5 : -.5));
```

```
rounded -> Array
(
    [Math.round(somenum)] => 21
    [~~ (0.5 + somenum)] => 21
    [~~ (somenum + (somenum > 0 ? .5 : -.5))] => 21
)
```

jQuery

What is jQuery?

jQuery is a fast and concise JavaScript Library created by John Resig in 2006 with a nice motto: Write less, do more. jQuery simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development.

jQuery is a JavaScript toolkit designed to simplify various tasks by writing less code. Here is the list of important core features supported by jQuery:
- DOM manipulation: The jQuery made it easy to select DOM elements, traverse them and modifying their content by using cross-browser open source selector engine called Sizzle.
- Event handling: The jQuery offers an elegant way to capture a wide variety of events, such as a user clicking on a link, without the need to clutter the HTML code itself with event handlers.
- AJAX Support: The jQuery helps you a lot to develop a responsive and feature-rich site using AJAX technology.
- Animations: The jQuery comes with plenty of built-in animation effects which you can use in your websites.
- Lightweight: The jQuery is very lightweight library - about 19KB in size (Minified and gzipped).
- Cross Browser Support: The jQuery has cross-browser support, and works well in IE 6.0+, FF 2.0+, Safari 3.0+, Chrome and Opera 9.0+
- Latest Technology: The jQuery supports CSS3 selectors and basic XPath syntax.

```
// Typically how you will typically call a function using jQuery.
$(document).ready(function(){});

// Doing the following will not invoke mymethod as a method but as a function.
// Because the callback is inside jQuery the function is bound to the document hence $(document) and that's what this refers to. If it wasn't jQuery the method would bound to the global window object (context) which means the variable this would be window.
$(document).ready(myobject.mymethod);

// Don't do this because it defeats the purpose of using jQuery's ready method because mymethod is invoked before the document is ready.
$(document).ready(myobject.mymethod());

// Invoke methods from inside the anonymous function because they typically reference object variables using the variable this.
$(document).ready(function(){myobject.mymethod()});

// This can also be solved by defining a bind method like this:

function bind(fnThis, fn){
    var args = Array.prototype.slice.call(arguments, 2);
    return function(){
        if (0 < args.length) arguments = args;
```

```
        return fn.apply(fnThis, arguments);
    };
}

$(document).ready(bind(myobject, myobject.mymethod));

// Sample Code Demonstrating This Behavior

var scope = "Global Scope";
var myobject = {
    scope: "Local Scope",
    mymethod: function(){
        alert(this.scope);
    }
};

// Undefined (there isn't a scope property in the document object)
$(document).ready(myobject.mymethod);

// Global Scope
$(document).ready(myobject.mymethod.bind(window));
$(document).ready($.proxy(myobject.mymethod, window));

// Local Scope
$(document).ready(function(){myobject.mymethod()});
$(document).ready(myobject.mymethod.bind(myobject));
$(document).ready(bind(myobject, myobject.mymethod));
$(document).ready($.proxy(myobject, "mymethod"));
$(document).ready($.proxy(myobject.mymethod, myobject));
```

$.noConflict()

When you put jQuery into no-conflict mode, you have the option of assigning a variable name to replace $.

You can continue to use the standard $ by wrapping your code in a self-executing anonymous function; this is a standard pattern for plugin authoring, where the author cannot know whether another library will have taken over the $.

However, the handler passed to the .ready() method can take an argument, which is passed the global jQuery object.

```
var $j = jQuery.noConflict();

// OR

(function($){
    // your code here, using the $
})(jQuery);

// OR

jQuery(document).ready(function($){
    // code using $ as usual goes here.
});
```

Detach Elements to Work With Them

The DOM is slow; you want to avoid manipulating it as much as possible. jQuery introduced $.fn.detach in version 1.4 to help address this issue, allowing you to remove an element from the DOM while you work with it.

```
var $table = $('#myTable');
var $parent = $table.parent();

$table.detach();
// ... add lots and lots of rows to table
$parent.append(table);
```

Don't Act on Absent Elements

jQuery won't tell you if you're trying to run a whole lot of code on an empty selection - it will proceed as though nothing's wrong. It's up to you to verify that your selection contains some elements.

You could make a plugin to use to ensure that the jQuery object is not empty.

```
// BAD: this runs three functions
// before it realizes there's nothing
// in the selection
$("#nosuchthing").slideUp();

// Better
var $mySelection = $("#nosuchthing");
if ($mySelection.length) { $mySelection.slideUp(); }

// BEST: add a doOnce plugin
jQuery.fn.doOnce = function(func){
    this.length && func.apply(this);
    return this;
};

$("li.cartitems").doOnce(function(){
    // make it ajax! \o/
});

// OR

$.fn.ensure = function(){
    if (this.length === 0) throw "Empty jQuery result.";
    return this;
};

$("ul.some-list").ensure().append(listItem);
```

This guidance is especially applicable for jQuery UI widgets, which have a lot of overhead even when the selection doesn't contain elements.

Selector Performance:
http://jsperf.com/wtfasdasdasd/2 (http://jsperf.com/wtfasdasdasd/2)
http://jsperf.com/wtfasdasdasd/5 (http://jsperf.com/wtfasdasdasd/5)
http://jsperf.com/find-vs-sizzle-jf (http://jsperf.com/find-vs-sizzle-jf)
http://jsperf.com/specific-left-or-right (http://jsperf.com/specific-left-or-right)

Event Listeners

JavaScript 1.8.5 (https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.8.5) introduces the Function.prototype.bind() (https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/bind) method, which lets you specify the value that should be used as this for all calls to a given function. This lets you easily bypass problems where it's unclear what this will be, depending on the context from which your function was called. Note, however, that you'll need to keep a reference to the listener around so you can later remove it.

addEventListener

addEventListener() registers a single event listener on a single target. The event target may be a single element in a document, the document itself, a window, or an XMLHttpRequest.

To register more than one event listener for the target, call addEventListener() (https://developer.mozilla.org/en-US/docs/DOM/EventTarget.addEventListener) for the same target but with different event types, event listeners or capture parameters.

Syntax

target.addEventListener(type, listener[, useCapture]);

type

A string representing the event type to listen for.

listener

The object that receives a notification when an event of the specified type occurs. This must be an object implementing the EventListener interface, or simply a JavaScript function.

useCapture (optional)

If true, useCapture indicates that the user wishes to initiate capture. After initiating capture, all events of the specified type will be dispatched to the registered listener before being dispatched to any EventTarget beneath it in the DOM tree. Events which are bubbling upward through the tree will not trigger a listener designated to use capture. Note that this parameter is not optional in all browser versions. If not specified, useCapture is false.

```
document.addEventListener("DOMContentLoaded", function(){
    alert("Document Ready");
});

function DOMContentLoaded(){
    alert("Document Ready");
}

document.addEventListener("DOMContentLoaded", DOMContentLoaded, false);
document.removeEventListener("DOMContentLoaded", DOMContentLoaded, false);
window.addEventListener("load", DOMContentLoaded, false);
window.attachEvent("onload", DOMContentLoaded);
document.attachEvent("onreadystatechange", DOMContentLoaded);
document.detachEvent("onreadystatechange", DOMContentLoaded);
```

readyState

The readyState property returns the (loading) status of the current document.

This property returns one of four values:
- uninitialized: has not started loading yet
- loading: loading
- interactive: has loaded enough and the user can interact with it
- complete: fully loaded

document.onreadystatechange is typically more optimal than window.onload as it fires before all external resources such as images are loaded.

document.readystate is a property and document.onload is an event.

```
// alternative to DOMContentLoaded
document.onreadystatechange = function(){
    if (document.readyState == "interactive") {
        initApplication();
    }
};

// alternative to load event
document.onreadystatechange = function(){
    if (document.readyState == "complete") {
        initApplication();
    }
};
```

The value of this within the handler

It is often desirable to reference the element from which the event handler was fired, such as when using a generic handler for a series of similar elements. When attaching a function using addEventListener() the value of this is changed-note that the value of this is passed to a function from the caller.

```
var el = document.getElementById("t");
el.addEventListener("click", function(){ modifyText(); }, false);
```

In the example above, the value of this within modifyText() when called from the click event is a reference to the table 't'. This is in contrast to the behavior that occurs if the handler is added in the HTML source:

```
<table id="t" onclick="modifyText();">
```

The value of this (http://www.quirksmode.org/js/this.html) within modifyText() when called from the onclick event will be a reference to the global (window) object.

Using a special function called handleEvent to catch any events:

```
var Something = function(element){
    this.name = 'Something Good';
    this.handleEvent = function(event){
        console.log(this.name); // 'Something Good', as this is the Something object
        switch (event.type) {
            case 'click':
                // some code here...
            break;
            case 'dblclick':
                // some code here...
```

```
            break;
        }
    };

    // Note that the listeners in this case are this, not this.handleEvent
    element.addEventListener('click', this, false);
    element.addEventListener('dblclick', this, false);

    // You can properly remove the listners
    element.removeEventListener('click', this, false);
    element.removeEventListener('dblclick', this, false);
};
```

Legacy Internet Explorer and attachEvent

In Internet Explorer versions prior to IE 9, you have to use attachEvent rather than the standard addEventListener. There is a drawback to attachEvent, the value of this will be a reference to the window object instead of the element on which it was fired.

```
if (el.addEventListener) {
    el.addEventListener('click', modifyText, false);
} else if (el.attachEvent) {
    el.attachEvent('onclick', modifyText);
}
```

Old way to add event handlers

addEventListener() was introduced with the DOM 2 Events specification. Before then, event listeners were registered as follows:

```
// Pass a function reference – do not add '()' after it, which would call the function!
el.onclick = modifyText;

// Using a function expression
el.onclick = function(){
    alert("Element Clicked!");
};
```

This method replaces the existing click event listener(s) on the element if there are any. Similarly for other events and associated event handlers such as blur (onblur), keypress (onkeypress), and so on.

Because it was essentially part of DOM 0, this method is very widely supported and requires no special cross-browser code; hence it is normally used to register event listeners dynamically unless the extra features of addEventListener() are needed.

Memory Issues

In the first case, a new (anonymous) function is created at each loop turn. In the second case, the same previously declared function is used as an event handler. This results in smaller memory consumption.

Moreover, in the first case, since no reference to the anonymous functions is kept, it is not possible to call

element.removeEventListener because we do not have a reference to the handler, while in the second case, it's possible to do myElement.removeEventListener("click", processEvent, false).

```
var i, els = document.getElementsByTagName('*');

// Case 1
for (i = 0; i < els.length; i++) {
    els[i].addEventListener("click", function(e){ /* do something */ }, false);
}

// Case 2
function processEvent(e){
    /* do something */
}

for (i = 0; i < els.length; i++) {
    els[i].addEventListener("click", processEvent, false);
}
```

Document Ready

Sample function to determine when the document has been loaded. Also sample script to calculate the time elapsed to load the document.

ContentLoaded (https://github.com/dperini/ContentLoaded/blob/master/src/contentloaded.js)
jQuery's Document Ready (http://jsfiddle.net/gerst20051/Za8fr/)

```
function contentLoaded(win, fn){
    var doc = win.document, done = false,
        add = doc.addEventListener ? 'addEventListener' : 'attachEvent',
        rem = doc.addEventListener ? 'removeEventListener' : 'detachEvent',
        pre = doc.addEventListener ? '' : 'on',
        init = function(){
            if (e.type == 'readystatechange' && doc.readyState != 'complete') return;
            (e.type == 'load' ? win : doc)[rem](pre + e.type, init, false);
            if (!done && (done = true)) fn.call(win, e.type || e);
        };
    if (doc.readyState == 'complete') fn.call(win, 'lazy');
    else {
        doc[add](pre + 'DOMContentLoaded', init, false);
        doc[add](pre + 'readystatechange', init, false);
        win[add](pre + 'load', init, false);
    }
}

var $tt = (new Date).getTime();
function timeElapsed(t){ return ((new Date()).getTime() - t); }

contentLoaded(window,
    function(e){
        window.status = window.defaultStatus =
            ' * ' + (e.type || e) + ' ' +
            ' - ' + (e.eventType ? e.eventType : 'native') +
            ' in ' + timeElapsed($tt) + ' ms.';
    }
```

);

CSS Style Manipulation

First, rely on the cascade whenever possible. CSS has a natural system of fallbacks built right in. Browsers take into account the last value that they were able to understand (this is how the cascade was designed to work). This means that if you order different solutions to the same problem from least advanced to most advanced, browsers will naturally use the most advanced solution they are capable of understanding.

Just like normal property fallbacks, vendor prefixes should be ordered from oldest version to newest so that each browser gets the best code it is capable of handling. If there is a standard syntax, you want to put that last so that as support for the standard increases, more and more browsers will use the best code.

Be aware that setAttribute will remove all other style properties that may already have been defined in the element's style object. If the some-element element above had an in-line style attribute of say style="font-size: 18px", that value would have been removed by the use of setAttribute.

http://www.stubbornella.org/content/2012/05/02/cross-browser-debugging-css/
(http://www.stubbornella.org/content/2012/05/02/cross-browser-debugging-css/)

DOM Manipulation

http://jsperf.com/childnodes-vs-firstchild (http://jsperf.com/childnodes-vs-firstchild)
http://jsperf.com/document-vs-getelementsbytagname (http://jsperf.com/document-vs-getelementsbytagname)

Modify Node Values

```
function modifyText(){
    var t1 = document.getElementById("t1");
    if (t1.firstChild.nodeValue == "three") {
        t1.firstChild.nodeValue = "two";
    } else {
        t1.firstChild.nodeValue = "three";
    }
}

// OR

function modifyText2(text){
    var t1 = document.getElementById("t2");
    t1.firstChild.nodeValue = text;
}

var pclass = document.querySelector("p.class");
pclasss.innerText = "This is a paragraph!";

var node = document.getElementById("myList2").lastChild;
var list = document.getElementById("myList1");
list.insertBefore(node, list.childNodes[0]);

keys(document);
values(document);

// document functions
```

```
addEvenetListener
adoptNode
appendChild
attributes
bgColor
captureEvents
clear
cloneNode
contains
createAttribute
createAttributeNS
createCDATASection
createComment
createDocumentFragment
createElement
createEvent
createExpression
createTextNode
createTreeWalker
dispatchEvent
fgColor
getCSSCanvasContext
getElementById
getElementsByClassName
getElementsByName
getElementsByTagName
getOverrideStyle
getSelection
hasAttributes
hasChildNodes
hasFocus
height
insertBefore
isPrototypeOf
linkColor
querySelector
querySelectorAll
readyState
referrer
releaseEvents
removeEventListener
title
valueOf
vlinkColor
width

// document elements
activeElement
all
anchors
applets
body
childNodes
documentElement
embeds
firstChild
forms
head
images
```

```
lastChild
links
nextSibling
nodeName
nodeType
nodeValue
parentElement
parentNode
previousSibling
removeChild
replaceChild
scripts
styleSheets
textContent
plugins

// document action events
onabort
onbeforecopy
onbeforecut
onbeforepaste
onblur
onchange
onclick
oncontextmenu
oncopy
oncut
ondblclick
ondrag
ondragend
ondragenter
ondragleave
ondragover
ondragstart
ondrop
onerror
onfocus
oninput
oninvalid
onkeydown
onkeyup
onload
onmousedown
onmousemove
onmouseout
onmouseover
onmouseup
onmousewheel
onpaste
onreadystatechange
onreset
onscroll
onsearch
onselect
onselectionchange
onselectstart
onsubmit

// checkout
elementFromPoint
```

evaluate
execCommand
importNode
isEqualNode
isSameNode
isSupported
lastModified
normalize
open
ownerDocument

document.documentElement

Read-only and returns the element that is the root element of the document (for example, the <html> element for HTML documents).

```
var rootElement = document.documentElement;
var firstTier = rootElement.childNodes;

// firstTier is the NodeList of the direct children of the root element
for (var i = 0; i < firstTier.length; i++) {
    // do something with each direct kid of the root element as firstTier[i]
}
```

This property is a read-only convenience for getting the root element associated with any document.

HTML documents typically contain a single child node, <html>, perhaps with a DOCTYPE declaration before it. XML documents often contain multiple child nodes: the root element, the DOCTYPE declaration, and processing instructions.

That's why you should use document.documentElement rather than document.firstChild to get the root element.

document.body

Returns the <body> or <frameset> node of the current document.

```
var objRef = document.body;
document.body = objRef;

// in HTML: <body id="oldBodyElement"></body>
alert(document.body.id); // "oldBodyElement"

var aNewBodyElement = document.createElement("body");

aNewBodyElement.id = "newBodyElement";
document.body = aNewBodyElement;
alert(document.body.id); // "newBodyElement"
```

document.body is the element that contains the content for the document. In documents with <body> contents, returns the <body> element, and in frameset documents, this returns the outermost <frameset> element.

Though body is settable, setting a new body on a document will effectively remove all the current children of the existing <body> element.

JavaScript CSS Selector Engine Timeline

2003.03.25: document.getElementsBySelector() (http://simonwillison.net/2003/Mar/25/getElementsBySelector/) by Simon Willison (later used in behaviour.js) [source (http://simonwillison.net/static/2003/getElementsBySelector.js)]
2004.04.10: CssQuery() 1.0 (http://dean.edwards.name/my/cssQuery/1_0/): by Dean Edwards [source (http://dean.edwards.name/my/cssQuery/1_0/cssQuery-source.js)]
2005.08.19: CssQuery() 2.0 (http://dean.edwards.name/my/cssQuery/). [source (http://dev.fckeditor.net/browser/FCKtest/runners/selenium/lib/cssQuery/src?rev=1044)]
2005.08.22: jSelect (http://ejohn.org/blog/selectors-in-javascript/) (precursor to jQuery) [source (http://ejohn.org/apps/jselect/selector.js)]
2006.01.14: jQuery (http://ejohn.org/blog/barcampnyc-wrap-up/) first release. [source (https://github.com/jquery/jquery/blob/1.0a/jquery/jquery.js#L526-606)]
2006.01.18: Prototype (http://ajaxian.com/archives/prototype-adds-css-selector-function-divpage-psummary-img). Initial release of selector engine. [source (https://github.com/sstephenson/prototype/blob/bb4d189b37b196dcd8bb0b5cb551e1cd7084596f/src/selector.js)]
2006.04.04: moo.dom (http://mad4milk.net/entry/moo.dom-easily-target-html-elements) (precursor to mootools) [source (http://moodom.mad4milk.net/moo.dom.js)]
2006.08.26: jQuery 1.0 (http://jquery.com/blog/2006/08/26/jquery-10/) [source (http://dev.jquery.com/browser/tags/1.0/src/jquery/jquery.js#L1157)]
2006.11.14: Mochikit Selector (http://www.mochikit.com/doc/html/MochiKit/Selector.html). (orig. ported from prototype) [source (http://trac.mochikit.com/browser/mochikit/trunk/MochiKit/Selector.js)]
2007.01.08: jQuery 1.1a (http://jquery.com/blog/2007/01/08/jquery-11a/) ("10-20x faster than 1.0") [source (http://dev.jquery.com/browser/tags/1.1a/src/selector/selector.js)]
2007.01.11: DomQuery (http://www.jackslocum.com/blog/2007/01/11/domquery-css-selector-basic-xpath-implementation-with-benchmarks/) by Jack Slocum (ExtJS). [source (http://www.yui-ext.com/deploy/ext-1.0.1/source/core/DomQuery.js)]
2007.02.05: dojo.query() (http://dojotoolkit.org/node/336). [source (http://trac.dojotoolkit.org/browser/trunk/src/query.js)]
2007.03.21: base2.DOM (http://dean.edwards.name/weblog/2007/03/yet-another/). [source (http://base2.googlecode.com/svn/version/1.0(beta2)/src/base2-dom.js)]
2007.05.01: Prototype 1.5.1 (http://www.prototypejs.org/2007/5/1/prototype-1-5-1-released) [source (http://dev.rubyonrails.org/browser/spinoffs/prototype/tags/rel_1-5-1/src/selector.js)]
2007.05.07: Mootools 1.1 (http://blog.mootools.net/2007/6/11/selectors-on-fire) [source (http://dev.mootools.net/browser/tags/1-10/Element/Element.Selectors.js)]
2007.07.01: jQuery 1.1.3 (http://jquery.com/blog/2007/07/01/jquery-113-800-faster-still-20kb/) ("800% faster") [source (http://dev.jquery.com/browser/tags/1.1.3.1/src/selector/selector.js)]
2007.07.10: Ext 1.1 RC1 (http://extjs.com/blog/2007/07/10/css-selectors-speed-myths/) [source (http://trac.pagodacms.org/browser/pagoda/misc/ajax_form/javascript/ext-1.1-rc1/source/core/DomQuery.js?rev=467)]
2007.07.10: Dojo 0.9 [source (http://download.dojotoolkit.org/release-0.9.0/dojo-release-0.9.0-src/dojo/_base/query.js)]
2007.12.04: YUI 2.4.0 (http://yuiblog.com/blog/2007/12/04/yuii-240/) [source (http://yui.yahooapis.com/2.4.1/build/selector/selector-beta-debug.js)]
2007.12.17: NWMatcher (http://javascript.nwbox.com/NWMatcher/) by Diego Perini [source (http://nwevents.googlecode.com/svn/trunk/nwmatcher.js)]
2007.12.17: DOMAssistant 2.5 (http://www.robertnyman.com/2007/12/17/domassistant-25-released-css-selector-support-new-ajax-methods-and-more-goodies-added/) by Robert Nyman [source (http://domassistant.googlecode.com/svn/branches/2.5/DOMAssistantComplete.js)]

Browser Quirks

Use insertBefore instead of appendChild to circumvent an IE6 bug.

http://jsperf.com/string-vs-regex/3 (http://jsperf.com/string-vs-regex/3)
if (readyState == "loaded" || readyState == "complete") {}
is much faster than
if (/loaded|complete/.test(readyState)) {}

When eval('{"key" : 42}') is called, { is interpreted as a block of code instead of an object literal. Hence, the Grouping Operator (parentheses) is used to force eval to interpret the JSON as an object literal: eval('({"key" : 42})');.

IE fires both onload and onreadystatechange in this method.

```
var s = document.createElement("script"),
    h = document.head || document.getElementsByTagName("head")[0] || document.documentElement,
    done = false;
s.src = "http://code.jquery.com/jquery.min.js (http://code.jquery.com/jquery.min.js)";
s.onload = s.onreadystatechange = function(){
    if (!this.readyState || this.readyState == "complete" || this.readyState == "loaded") {
        if (!done && (done=true)) {
            main();
            s.onload = s.onreadystatechange = null;
            if (h && s.parentNode) h.removeChild(s);
            s = undefined;
        }
    }
};
h.insertBefore(s, h.firstChild);
```

Array.prototype.slice.call

IE 6 does not support this technique on nodeLists (childNodes and lists returned by getElementsByTagName()). In fact it will break you JavaScript code and throw an exception.

```
// arguments
function getArgumentsAsArray(){
    return Array.prototype.slice.call(arguments);
}

// nodelist
function getElementsByTagName(el, tagName){
    return Array.prototype.slice.call(el.getElementsByTagName(tagName));
}
```

Although ECMAScript makes iteration order of objects implementation-dependent, it may appear that all major browsers support an iteration order based on the earliest added property coming first (at least for properties not on the prototype). However, in the case of Internet Explorer, when one uses delete on a property, some confusing behavior results, preventing other browsers from using simple objects like object literals as ordered associative arrays. In Explorer, while the property value is indeed set to undefined, if one later adds back a property with the same name, the property will be iterated in its old position--not at the end of the iteration sequence as one might expect after having deleted the property and then added it back.

So if you want to simulate an ordered associative array in a cross-browser environment, you are forced to either use two separate arrays (one for the keys and the other for the values), or build an array of single-property objects, etc.

Can't write to {}.__proto__ in IE.

XHR

Microsoft failed to properly implement the XMLHttpRequest in IE7 (can't request local files), so use the ActiveXObject when it is available. Additionally XMLHttpRequest can be disabled in IE7/IE8 so you need a fallback.

Add protocol if not provided (IE7 issue with protocol-less urls.

Firefox throws exceptions when accessing properties of an xhr when a network error occurred.

When requesting binary data, IE6-9 will throw an exception on any attempt to access responseText.

Firefox throws an exception when accessing statusText for faulty cross-domain requests.

IE - response.text sometimes returns 1223 when it should be 204.

A bunch of bugs in cloning input elements in IE:

IE6-8 fails to persist the checked state of a cloned checkbox or radio button. Worse, IE6-7 fail to give the cloned element a checked appearance if the defaultChecked value isn't also set.

IE6-7 get confused and end up setting the value of a cloned checkbox/radio button to an empty string instead of "on".

IE6-8 fails to set the defaultValue to the correct value when cloning other types of input fields.

```
JavaScript:
document.getElementById("obj").getElementsByTagName("*");

HTML:
<object id="obj">
<param name="src" value="test.mov"/>
<param name="title" value="My Video"/>
</object>
```

Common CSS Tweaks in IE

- Needing to add hasLayout with zoom:1
- Position relative causing things to disappear
- 3px float bug
- Expanding container float bug (useful!) and overflow hidden which unfortunately "fixes" this useful bug.

Undeclared assignment throws in IE, when identifier corresponds to element name/id. jsFiddle

(http://jsfiddle.net/gerst20051/cgAds/)

So, you think you know JavaScript?

1. Scope

What value gets alerted?

```
var foo = 1;
function bar(){
    if (!foo) {
        var foo = 10;
    }
    alert(foo);
}
bar();
```

2. Scope / Function Hoisting and Returning

What value gets alerted?

```
var a = 1;
function b(){
    a = 10;
    return;
    function a(){}
}
b();
alert(a);
```

3. Scope

What value gets alerted?

```
if (!("a" in window)) {
    var a = 1;
}
alert(a);
```

4. Scope and Variable Initialization

What value gets alerted?

```
if (!("a" in window)) {
    a = 1;
}
alert(a);
```

5. Variable and Function Names

What value gets alerted?

```
var a = 1,
    b = function a(x){
        x && a(--x);
    };
alert(a);
```

6. Variable Initialization

What value gets alerted?

```
function a(x){
    return x * 2;
}
var a;
alert(a);
```

7. Modifying Arguments Variable

What value gets alerted?

```
function b(x, y, a){
    arguments[2] = 10;
    alert(a);
}
b(1, 2, 3);
```

8. Context Using Null

What value gets alerted?

```
function a(){
    alert(this);
}
a.call(null);
```

9. Closure

What value do you get for baz, bim, and bar when accessed outside the function?

```
// a self-executing anonymous function

(function(){
    var baz = 1;
    var bim = function(){ alert(baz); };
    bar = function(){ alert(baz); };
})();

console.log(baz);
bim();
bar();
```

10. Primitive Objects

Will found evaluate to true or false? If found is true "Found" will be alerted.

```
var found = new Boolean(false);

if (found) {
    alert("Found");
}
```

11. Hoisting Duplicates Behavior

What values gets logged for (typeof f) and for (f())?

```
console.log(typeof f);
if (true) {
    function f(){ return 1; }
} else {
    function f(){ return 2; }
}
console.log(f());
```

12. Math.min / Math.max

Does this expression evaluate to true or false?

```
console.log(Math.min() < Math.max());
```

13. Hoisting / Multiple Function Statements

What value does the first bar() return? What value does the second bar() return? What value does the last bar() return?

```
function bar(){ return 1; }
bar();
if (true) {
    // overwritting with function statement
    function bar(){ return 2; }
    bar();
}
function bar(){ return 3; }
bar();
```

#. Topic

Question?

Quiz Answers

1. 10

The browser will alert 10. JavaScript isn't block scoped but function scoped. The var foo inside the conditional brackets causes foo to be undefined (a falsy value) which causes the conditional to evaluate to true; therefore, the value of foo is set to 10. As we already know, all functions and variables are hoisted to the top of it's scope, but aren't initialized as demonstrated in the following code.

It looks like this.

```
var foo = 1;
function bar(){
    var foo;
    if (!foo) {
        foo = 10;
    }
    alert(foo);
}
bar();
```

2. 1

Here, of course, the browser will alert "1". So what's going on here? While it might seem strange, dangerous, and confusing, this is actually a powerful and expressive feature of the language. I don't know if there is a standard name for this specific behavior, but I've come to like the term "hoisting".

It looks like this.

```
var a;
```

```
function b(){
    var a = function(){};
    a = 10;
    return;
}
a = 1;
b();
alert(a);
```

3. undefined

4. 1

5. 1

6. function a(x){ ...

```
"function a(x){
        return x * 2;
}"
```

7. 3

8. window

9. error, error, 1

```
baz // ReferenceError: baz is not defined
bim // ReferenceError: bim is not defined
bar // 1
```

baz and bim are not defined outside of the function, so they will result in a ReferenceError.

bar is defined outside of the anonymous function because it wasn't declared with var; furthermore, because it was defined in the same scope as baz, it has access to baz even though other code outside of the function does not.

10. true

Found is alerted. In this example, a Boolean object is given a value of false, yet console.log("Found") still executes. That's because an object is always coerced to true when placed in a conditional statement. It doesn't matter that the object represents false, only that it is an object, and therefore evaluates to true.

11. function, 2

12. true

The start number (used in comparisons) for Math.min should be Infinity. All number that are lower than positive infinity should be the smallest from a list, if there are no smaller then Infinity is the smallest (the only option) in the list. And for Math.max it's the same; all numbers that are larger than negative infinity should be the biggest if there are no bigger.

Math.min(5, 100) where 5 is smaller than positive infinity (Infinity) and 100 it will return 5.

Calling Math.min and Math.max with an array parameter may won't work on every platform. You should do the following instead:

```
Math.min.apply(null, [1, 2, 3, 4, 5]); // returns 1
```

Where the first parameter is the scope argument. Because Math.min() and Math.max() are "static" functions, we do not give this scope argument a value.

If no arguments are given, the result is Infinity for Math.min() and -Infinity for Math.max(). If any value is

NaN, the result is NaN.

```
Math.max([1, 2, 3, 4]); // NaN
```

13. 3, 3, 3

Before the code even reaches the if statement the value of the variable bar is already set to the the last bar function (due to function hoisting) which returns 3.

#.

FAQ

What exactly is undefined?

Can you assign undefined to a variable?

You can certainly assign undefined to it, but that won't delete the variable. Only the delete object.property operator really removes things but delete is really meant for properties rather than variables as such. Browsers will let you get away with straight delete variable, but it's not a good idea and won't work in ECMAScript Fifth Edition's strict mode. If you want to free up a reference to something so it can be garbage-collected, it would be more usual to say variable = null.

What does if (!foo) actually do?

Why should you split script tag?

The first occurrence of the character sequence "</" (end-tag open delimiter) is treated as terminating the end of the element's content.

</script> has to be broken up because otherwise it would end the enclosing <script></script> block too early. Really it should be split between the < and the /, because a script block is supposed (according to SGML (http://www.w3.org/TR/html4/types.html#type-cdata)) to be terminated by any end-tag open (ETAGO (http://mathiasbynens.be/notes/etago)) sequence (i.e. </):

```
document.write("<script><\/script>");
// OR
document.write("<scr"+"ipt></sc"+"ript>");
// OR
document.write("\x3Cscript>\x3C/script>");
```

Whenever you need to use </style> inside a <style> element, or </script> inside a <script> element, just escape these strings. In both CSS and JavaScript there are various ways of doing this, but using a backslash (\, also known as "reverse solidus character") is by far the simplest.

Another method used to get around this is to dynamically create and insert a script tag.

```
var script = document.createElement('script');
script.src = 'http://ads.com/buyme?rand= (http://ads.com/buyme?rand=)'+Math.random();

// now append the script into HEAD, it will fetched and executed
document.documentElement.firstChild.appendChild(script);
```

Additional Reading / Watching

Here I provide additional references for you to continue your journey of mastering the best programming language in the world!

I highly recommend my JavaScript Guide (http://pastebin.com/uWc3DSCy) it will eventually be merged with this book.

I also wrote a condensed version of Douglas Crockford's already very dense book The Good Parts available here (http://pastebin.com/N0nE2xCA).

I'm working on a collection of various JavaScript pitfalls you should be aware of available here (http://jsfiddle.net/gerst20051/BfNa8/).

If you're interested in various JavaScript assertions check this jsfiddle (http://jsfiddle.net/gerst20051/MGz6g/) out.

Tests / Experiments

http://kangax.github.io/jstests/ (http://kangax.github.io/jstests/)

Recommended YouTube Videos

Crockford on JavaScript: YUI Theater (8 Video Playlist) (http://www.youtube.com/playlist?list=PL7664379246A246CB)
- The Early Years ()
- And Then There Was JavaScript ()
- Function The Ultimate ()
- The Metamorphosis of Ajax ()
- The End of All Things ()
- Loopage ()
- ECMAScript 5: The New Parts ()
- Programming Style & Your Brain ()

The JavaScript Trilogy by Douglas Crockford (3 Video Playlist) (http://www.youtube.com/playlist?list=PL5586336C26BDB324)
- Douglas Crockford: The JavaScript Programming Language (http://www.youtube.com/watch?v=v2ifWcnQs6M)
- In this 2007 presentation at Yahoo!, which is meant to be the beginning of a three-course sequence (followed by "Theory of the DOM" and then "Advanced JavaScript"), Douglas Crockford explores not only the language as it is today but also how the language came to be the way it is.
- Douglas Crockford: An Inconvenient API - The Theory of the DOM (http://www.youtube.com/watch?v=Y2Y0U-2qJMs)
- In an internal tech talk at Yahoo! in 2006, Douglas Crockford delves into the sordid history of the DOM: that "vast source of incompatibility, pain and misery" that frontend engineers love to hate.
- Douglas Crockford: Advanced JavaScript (http://www.youtube.com/watch?v=DwYPG6vreJg)
- In this presentation (the third of a three-part series) Douglas Crockford looks closely at code patterns from which JavaScript programmers can choose in authoring their applications. He compares familiar constructs like the Pseudoclassical Pattern with more unique patterns like the Parasitic Pattern that (he argues) run more "with the grain" of JavaScript.

Google I/O 2011: Learning to Love JavaScript (http://www.youtube.com/watch?v=seX7jYI96GE)
- Alex Russell. JavaScript remains one of the most popular and important programming languages in history. Web

Developer and Chrome Engineer Alex Russell exposes the timeless strengths of the JavaScript language and why it is a vital part of the open web platform. Come hear what's next for the JavaScript standard and how to get the most out of the new features coming soon in V8 and Chrome.
- Arrays are objects. Indexes get turned into strings and then referenced that way. Everything being an object also being a map.

JavaScript: The Good Parts (http://www.youtube.com/watch?v=hQVTIJBZook)
- Douglas Crockford. JavaScript is a language with more than its share of bad parts. It went from non-existence to global adoption in an alarmingly short period of time. It never had an interval in the lab when it could be tried out and polished. JavaScript has some extraordinarily good parts. In JavaScript there is a beautiful, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript was so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent abomination. This session will expose the goodness in JavaScript, an outstanding dynamic programming language. Within the language is an elegant subset that is vastly superior to the language as a whole, being more reliable, readable and maintainable.

Javascript: Your New Overlord (http://www.youtube.com/watch?v=Trurfqh_6fQ)
- In this keynote presentation at JAXConf 2012, Douglas "The JavaScript Guy" Crockford asserts that JavaScript has become the most important programming language in the world.
- "The first time I saw JavaScript in 1995 I thought this is the stupidest thing I've ever seen. I said that. And I was pretty confident that I was right. ... In looking deeply at it I discovered it's got lambdas in it! Suddenly all this potential opened up."

Dave Herman: The Future of JavaScript (http://www.youtube.com/watch?v=u4IdoBU1uKE)
- Dave Herman. Mozilla Labs engineer and TC39 representative Dave Herman joined us at YUIConf 2011 to give this keynote talk on the future of JavaScript, covering many of the new features currently under consideration for ES6, the next edition of the ECMAScript standard.

Nicholas Zakas: Scalable JavaScript Application Architecture (http://www.youtube.com/watch?v=vXjVFPosQHw)
- SlideShare (http://www.slideshare.net/nzakas/scalable-javascript-application-architecture)
- Nicholas Zakas. Yahoo! home page engineer Nicholas Zakas, author of Professional JavaScript for Web Developers, discusses frontend architecture for complex, modular web applications with significant JavaScript elements.

Fluent 2012: Brendan Eich, "JavaScript at 17" (http://www.youtube.com/watch?v=Rj49rmc01Hs)
- Brendan Eich. Almost two decades after the birth of JavaScript, its creator gives a whirlwind history of the language with stories (and dirt!) dished out from each era. What worked well for JavaScript and what has continued to make developers groan? What's coming in ES6 and where next for the JavaScript community? Answers to these questions and more from as authoritative a source as it gets.

Introduction to Node.js with Ryan Dahl (http://www.youtube.com/watch?v=jo_B4LTHi3I)
- Node.js is a system for building network services for Google's V8 JavaScript engine. In this presentation Ryan Dahl, the man behind Node.js will introduce you to this event-driven I/O framework with a few examples showing Node.js in action. Ryan will also talk about the recent release of v0.4.0 and how to use some of the new APIs.

Node.js: JavaScript on the Server (http://www.youtube.com/watch?v=F6k8lTrAE2g)
- Ryan Dahl. Presented by Ryan Dahl, the creator of the node.JS open source project.
- It is well known that event loops rather than threads are required for high-performance servers. Javascript is a language unencumbered of threads and designed specifically to be used with synchronous evented I/O, making it an attractive means of programming server software.
- Node.js ties together the V8 Javascript compiler with an event loop, a thread pool for making blocking system calls, and a carefully designed HTTP parser to provide a browser-like interface to creating fast server-side software. This talk will explain Node's design and how to get started with it.

Hands on with Node.js / Beginner Guide / Getting Started (http://www.youtube.com/watch?v=_l96hPlqzcI)
- Node.js is an exciting new platform for building web applications in JavaScript. With its unique I/O model, it excels at the sort of scalable and real-time situations we are increasingly demanding of our servers. The ability to use JavaScript for both the client and server opens up many possibilities for code sharing, expertise reuse, and rapid development. The class is intended for anyone looking to explore the capabilities of the Node.js development platform.
- At the end of the class, students will have gained a grasp on node's ecosystem and paradigms, be able to write node modules that they can publish and share, and will have experimented with writing realtime web applications.

```
Next Video ()
● Description
```

Thanks for reading my book, if you have any questions or suggestions feel free to email me at gerst20051@gmail.com