INTRODUCTION TO JAVASCRIPT FOR JAVA PROGRAMMMERS

a seminar by Ed de Luna

CLASS NOTES

HOW TO USE AND CODE IN JS

Executing code in JavaScript is one of easiest things to accomplish. There are a plethora of ways to approach JavaScript programming, but nothing beats its home turf: The Browser.

The fact that you are listening to this conference right now means that you are in the possession of a computer with a web browser, unless you are listening to this from a mobile device, in which case the following does not apply to you.

Please Open a New Tab or Window of your web browser, right click anywhere on the screen. On the sub menu, please click Inspect Element. In a moment, either at the bottom or the right-hand side a window will open, containing the markup of your new tab page. You will see menus labeled Web Inspector, Elements, Network and others. Click on the Button that says "Console". This is the debug console to which the JS compiler prints. Click directly on the console's white space and type console.log("Hello World"): and click enter.

Congratulations, you have just executed a JavaScript script. Every JavaScript Console can be used as an analog to Python's own Integrated Development Environment, running one line of code at a time. This is fine to experiment with a website that has already been loaded, modifying the data or automating tedious tasks within the page.

However, hacking the top layer of the front-end is just the beginning. To write your own JavaScript files and get the browser to execute them, you have to write an HTML file that references them. The scrip tag is the designated markup for that effect. You can write the scripts directly into the markup, or you can reference an external file that contains the instructions in the src attribute, which is the best practice. Putting the tag at different places changes with what information the JS script executes. The best practice is to put the script tag on the bottom of the body tag in HTML. New non-standard attributes, like async and deferred, change the way scripts are loaded and will become standard in the future. (Refer to example, best practices to put script tags)

It's important to note that it is possible to concatenate multiple script files and have a file reference content that has already been declared and initialized in another script file. This is due to the fact that the in-browser engine loads scripts one after the other and pools all the data in a single Virtual Machine. This will make more sense as time goes and to some it may sound like a massive security oversight, but this particular feature enables an incredible amount of extensibility and modularity to large and complex applications. (refer to example 2)

If you feel that what you are doing in JS isn't large or important enough to warrant its own markup page or you are simply messing around with the language, as most people do, loading an HTML file with

multiple scripts does feel like overkill. To this end, Google and Mozilla offer tools to write, save and execute code without the need to write HTML markup. JavaScript Editey for Google Chrome does just that: You write pure JS and let the browser execute it, with full access to console. Mozilla Aurora, the developer version of Firefox, offers such a feature out-of-the-box called Scratchpad.

To make things even simpler, there are multiple sites that allow users to write HTML, CSS, and JavaScript inside their own mainframes, execute it and see the results live, without the need of downloading special browsers or even storing anything locally. JSFiddle.net is such a platform, and you also have the ability to save and share these code snippets with the world for free. Codepen.io offers the same service, however its designed differently with web design in mind. Runnable.com offers a similar service, allowing you to create your own file tree and even run PHP stacks, Python stacks, Node.js stacks and a plethora of other web technologies for free. It does not get easier than this and even more services are coming down the pipeline to make programming on the web even easier.

For people with a bit more experience who wish to have code auto completion, linters, builders, and file management through services like Apache Ant, there are two alternatives: NetBeans WebStorm, a paid once of software that is by and far the best IDE for the web, and Sublime Text 3, which is my personal tool of choice. Sublime is an extensible, moddable, and flexible notepad program that stands out among text editors for its intuitive and clean design. Sublime Text 3 is technically shareware, but you can use the program for free almost indefinitely.

Apart from hardware costs, the setup costs to learn JS are zero. A plethora of tools, mods, tutorials, people, and community are available with a simple Google search and, like I mentioned before, if you have a web browser in a computer, you can execute JavaScript.

This almost chameleon versatility is possible thanks to JavaScript Engines, virtual machines that interpret and execute JavaScript code. Brendan Eich built the first JS Engine, Spidermonkey, at the same time he was designing JavaScript. Implemented in C++, at first the engine implemented Tracing Just-In-Time Compilation to optimize execution, recording control flow and data types, only to be replaced by a Whole-Method JIT Compiler. Today, SpiderMonkey powers Mozilla Firefox and Aurora.

However, the fastest JS Engine in existence right now by far is Lars Bak's V8, today implemented in Google Chrome. It compiles JS code to native machine code before execution instead of interpreting it into byte code, like the Java Compiler or the Spidermonkey engine. The code is optimize twice dynamically at runtime, making it the fastest on the world. Furthermore V8 was designed to break out of the browser space as a standalone, open-sourced engine that can be integrated into a multitude of projects, including a gem called Node.js that I will talk about later.

One Virtual Machine per Tab, per Window. The browser has come a long way and so has its programming language of choice.

Without further ado, let's dive into the syntax.

JAVASCRIPT SYNTAX ESSENTIALS

It's important to note from the get-go that, despite being named after it, JavaScript is not a subset of Java in any way, shape or form. The syntax and designed were inspired by Java, however that is where the similarities end. Due to the fact that the intended audience is familiar with the Java Programming Language, I will stop every so often to indicate where JavaScript departs from Java in significant and

profound ways. That being said, the best analogy I have found on the web is "JavaScript is to Java what car is to carpet." They share letters, and some key elements. However, they are not related.

(view example 3)
The JavaScript syntax looks like a hybrid between the reserved keywords of Java and the clarity and effectiveness of Python, and with good reason: Its HyperTalk roots caused the syntax to be simple as it is. JavaScript shares the same Unary, binary, arithmetic, comparison, logical and assignment operators as Java. Furthermore, identifiers in JavaScript are case sensitive for all data types, declaring variables and assigning them values just like any C-based language. The "+" sign is overloaded for both addition and string concatenation. Uninitialized identifiers hold an undefined value. Single-lined and Multi-lined comments work exactly like Java.

Statements are terminated with a semicolon as a best practice. I say "best practice" because the JavaScript Virtual Machine does not actually require them. Spaces, tabs, and new lines, in a parallel to Python, play a significant role in syntax. If a semicolon is missing, the Virtual Machine places the punctuation by itself where a statement is parseable and well formed. However, this may lead to unintended effects in runtime, causing two statements to run like one and other oddities. Because of this, it is recommended that every statement should be terminated with a semi-colon, and instead use whitespace and tabs as a way to space out the code and make human-readable.

The keen eye may have noticed an immediate departure from the usual fare of programming languages, if all you have ever done is ever Java or C: Variables are not declared by their type. This is because JavaScript implements duck typing: Any identifier can store any sort of variable. This can be hard to process and it requires a shift in thinking about computation. After all, how do you make sure a variable is of a certain type before trying to start an illegal operation, such as invoking a String method on a number? For such needs, JavaScript incorporates a typeof operand that outputs the type of a variable compared to another variable or its class name (example 4).

How is it possible to have a single variable initializer without indicating its type? It would sound absolutely insane to a person used to think in C and Java. The answer is incredibly simple: There is no such thing as a primitive type in JavaScript.

In JavaScript, in one way or another everything is an object.

There is no such thing as a primitive data type in JavaScript, or at least in the way we think about them. In fact there is no concept of integers, floats, or shorts. The distinction between Methods and Objects does not exist. The only thing that approaches what we think of as primitive data are numbers and boolean, and even numbers are debatable objects that box and un-box automatically much like the Primitive Wrapper Classes in Java. At any rate, JavaScript eschews the concept of distinguishing between two types of data and chooses to declare, initialize, and use them on the same manner.

In JavaScript, variables are function-scoped, rather than being block-scoped like traditional C-based languages. This means that, technically, you could make use of the iterator variable of a for loop outside of said for loop. What's more, if a variable is declared outside of a function scope, or without the "var" keyword, they are incorporated into the Global Object (more on that later), and placed on what could be called an equivalent to a public access modifier. These variables are not subject to order of initialization: You could add a number to a variable that comes after it. This is called variable hoisting. When referencing an identifier, the Virtual Machine goes through what you could call a reverse matryoshka: It checks the most immediate local variable scope and progressively traverses

broader and broader scopes until it checks the global object scope, where everything else is contained. If the Virtual Machine cannot find the identifier after traversing from the local to the global scope, the scope chain, then it's time to output an error. This is a huge departure from Java and others, where the scope of variables and references is tightly controlled through access modifiers, class fields, and static identifiers (example 5).

In JavaScript there is but one number type and that is Number, a 64-bit floating point, usually thought of as Double in other statically typed languages. In previous iterations, JavaScript had glitches in arithmetic computing, however this has been fixed in the Version 5 of the ECMA Script Standard.

When an invalid number operation takes place, such as a division of a number by a string, the value takes the form of NaN, a special number reserved for either undefined or erroneous operations. It propagates quickly, as any operation with NaN as a factor will have NaN as result. Oddly enough, one cannot check against this fallback value, as console.log(NaN === NaN) is false. This is not really helpful, as it doesn't indicate where exactly the computation would go wrong, as opposed to Java, which would return a runtime error. Remember, JavaScript was designed to run in the background, as such it's very silent about it's errors unless the developer really knows what he or she is doing. To parse a String into a Number, the Global Object provides a method called parseInt(). The Number() method also servers this function (example 6).

JavaScript provides a Boolean data type, expressed as true and false literals. Logical and comparison operators work identically in Java. The place of major departure consists in the interpretation of values belonging to other data types. Douglas Crockford describes them as "truthy" and "falsy" when some values are evaluated in the comparison and logical context. For example, if we type console.log(true == 0), we will receive a false statement. JavaScript will interpret as "false" the following: The number 0, empty strings, strings containing the number 0, and the null operator. NaN and undefined, the initializing value of all variables, are interpreted as neither true nor false. This is a quirk of the language that has creates a type checking problem for some algorithms and implementations. (example 7)

Strings are handled similarly to the Java implementation of these objects: They are sequences of UTF-16 code units, immutable, with string literals able to use single and double quotes for their declaration. Unlike Java, string comparison is done using the == operator. You can know the length of a string using the .length field. Parsing a Number or Array into a String is done using the String() method. Like Java, the String object in JavaScript has a number of methods that create a new, modified string. These methods include charAt, indexOf, match (for checking against a regular expression), slice, substring, toLowerCase and toUpperCase. If you are familiar with the String class in basically any C-based language, you should be able to know what these are. You can concatenate strings using the + operand or a concat() method if you want to.(example 8)

Arrays in JavaScript behave very differently from their Java counterparts. In fact they have more in common with the List Abstract Structure. For one, they are dynamic in size, with the user being able to push objects into an empty array. Due to the typing nature of variables in JS, arrays are able to hold multiple data types in one single structure. Like Java, their indexing scheme is zero based and return their size with the .length field. Array literals are declared using square brackets, with their elements separated by commas. They can also be declared through the constructor-esque method new Array(), with a predefined size. Furthermore, arrays have built in methods that modify its data: join(), to put all array elements into a string, slice(), in order to create a subset of an array, push() to place an element at the end of an array. (example 9)

Control Structures, such as if/else, try/catch, switch, do/while, while, and for loops are all identical to their Java counterparts. Bear in mind that due to the fact that variables are not scope-blocked, but function-blocked, you can easily get an unexpected result if you access variables declared and initialized outside their intended scope. Apart from that, there is no notable difference between the control structures of both language. It must be mentioned that JS supports loop and block labeling for use of the continue and break statements.

Functions are declared in a vastly different manner than their Java counterparts. We have already mentioned how the lack of access modifiers saddles the developer with managing the scope of their variables. However, this is true up to a point. There are several ways to declare a function in JavaScript: First, you can declare a function in the global namespace. Function declared in a global scope must have a name . Functions may or may not include a list of expected arguments and may or may not return something. Functions are, by default, void. It is also possible to store functions in a variable, give this variable a name and execute the variable anywhere within its scope (example 10).

Again, variables declared within the method are scoped within the method itself, rather than control flow structures, so any and all data that is created and used in the method is only accessible there. The argument list is an array-like object that can be referenced even when there are no expected arguments declared within the function.

You can put arguments in a method that doesn’t specify them, reference the arguments object as if it was an array and use that data. The arguments object includes a .length property, however it doesn’t have .splice(), .join() or other Array Functions. There is a lot more that can be said about the arguments which we will talk about later. Suffice it to say, arguments in JavaScript Functions work like an array (example 10).

Numbers, booleans, and strings are passed by value, while Objects and all other data types are passed in arguments by reference. Functions themselves can pass as arguments of other functions, you can even declare, execute functions within functions, and even call them outside of the scope where they were initialized if the variable containing the function is declared outside of it (example 10). It's also possible to call a function and write an anonymous function as an argument, without the need of giving it a name or identifying it. The reason why is this possible is that functions themselves are first-class objects in the JS Programming Language. Again, everything is an object or based around objects. Because, functions themselves have methods of their own. These are .bind(), .apply(), and .call(), for now methods that escape the scope of this seminar.

The object is the backbone of JavaScript, its alpha and omega, and the greatest departure from the Java programming language and what it grants it its almost chameleon nature. As I mentioned earlier, there are no classes, interfaces, or abstracts in JavaScript. This is the one of the most difficult concepts to understand as a developer who has worked in traditional object orientation, because it requires a different way of thinking. At first, you might find the way JS approaches objects to be counter-intuitive, but bear with me. Instead of the traditional class think about JS Objects in terms of associative hashes or dictionaries as implemented in Python.

Objects are declared using the built-in constructor new Object, or with the Object Literal, which is two empty curly braces. Both of these methods create an empty Object. It's important to note that, unlike Java, declaring two variables with the object literal do not result in two variables pointing at the same object. All literals create objects independent objects from one another. An object holds reference names, which could correspond to functions, numbers, booleans, arrays, strings, or other objects. Like

Java, they can be accessed and assigned using dot notation. You can also access fields using an array-esque notation, putting the name of the object, a pair of square brackets, and the name of the object field inside a string. The Engine will interpret this and return the field you are looking for. Furthermore, using the object literal you can initialize the object itself with all the necessary properties and methods the user decides, right from the get-go. Inside the object literal, variables are declared without the var keyword. Instead variables are initialized with a colon (:). Each variable is separated from each other with a comma, akin to initializing a traditional array(example 11).

/*The JavaScript Object Notation is based on the simplicity of JavaScript object literals. They are simple associative arrays, capable of containing all data types, with the exception of functions. Declaring and initializing them is the same process as regular JS objects. However, once the data is ready to be transmitted, it's necessary to convert it to a string. The JSON global object contains a method called JSON.stringify() to do so. In order to parse a stringified JSON, received from another process or application, JSON.parse() is used. (example 11)*/

Objects are not tied down to static classes that determine what kind of data they can hold. All fields created within the local scope of the object are limited to that scope and others that may exist inside. However, what if I have two variables declared with the same name in two different scopes? What if the name of a variable conflicts directly with a global constant? How can we reconcile the lack of access modifiers with proper encapsulation? Eich responded to this conundrum with the "this" object. This serves as a way to distinguish between locally declared variables and fields belonging to a larger scope. When a function is executed, it gets the this property -- a variable with the value of the object that invokes the function where this is used. Because of this, the 'this' object contains the value of the object that invoked the function, a shortcut to the invoker of the function.
As we go deeper in the scope chain, the value of 'this' becomes handy in accessing methods and variables that would normally be outside of scope. Observe example 12's execution. In global scope, "this" refers to the window object, the main container of all loaded scripts (example 12).

I have been mentioning something called a Global Object throughout this session. We touched on it briefly, the object "this" refers when it is not encapsulated by any sort of function or object. The global object is a regular JavaScript object that serves a very important purpose: the properties of this object are the globally defined symbols that are available to a JavaScript program. When the JavaScript interpreter starts (or whenever a web browser loads a new page), it creates a new global object and gives it an initial set of properties that defines global properties, global functions, constructor functions like Date(), String(), Object(): and global objects like the JSON object we discussed a while ago. JavaScript Engines encapsulate all variables, objects, and methods inside the global object. All the methods, variables, and objects from different script files go into the same global. When you declare variables without the var keyword and functions without a variable container, you initialize them into the global object. In browsers, the global object makes reference to an alternate name: window. This object has additional properties, such as location, outerWidth, sidebar, among others. However this is not always the case, as you may find JavaScript where there is no web browser window. However, all execution happens within the Global Object, where all the variables are widely accessible and modifiable by all possible scopes. This may seem like a good thing, but like public fields in Java objects, it's not a good idea to leave everything unencapsulated. Globally-declared variables have a problem when they are accidentally changed , creating errors that propagate in the execution. To remedy this, a good common practice is to namespace all objects within a custom object. By declaring everything within the object literal, or adding to it using dot notation, you ensure that everything in the context of execution will be limited to a scope of the developer's own choosing, rather than being vulnerable to the troubles of the global scope. To name this namespace, it's recommended to use a

unique and consistent name in order to maintain code portability. If the namespaced code is spread out in multiple script files, it's important to always have a fallback should the namespacing object not exist. it's quite simple:

/*var myNamespace = myNameSpace || {}.*/

By doing this, the developer secures the data and allows external libraries to play a role in one's own scripts. Namespace increases the portability of JavaScript code considerably, making it cleaner and safer to execute (example 13).

For further information, do not hesitate to check Crockford on JavaScript's seminar on the YUI Theater called The JavaScript Programming Language.

PROTOTYPAL INHERITANCE

JavaScript's greatest departure from Java is the way that it handles inheritance. As stated before, the concept of Classes, Interfaces, and Abstracts does not exist within the syntax. To people who have learned Object Oriented Programming revolving around these concepts, the lack of these concepts is confusing. To fully understand how important this is, let's recap the basics of class-based inheritance quickly:

Let us say you have a series of classes that execute similar tasks, have similar or even identical fields and methods. However these classes are distinct from each other enough to be considered separate. In order to to prevent having to rewrite similar methods and declare similar fields over and over again, making for some incredibly messy code, a parental class is created in an IS-A relationship scheme. This parental class abstracts the common and repeated code among all the classes, making our previous collection of messy classes free of it. In turn. the now child classes extend the parental class in a one-to-one relation. A class cannot have multiple parents, but a class can have any amount of child classes that extend it. The child classes then serve as a blueprint to instantiate objects that also hold this relation. Parental classes that merely abstract code and don't have a use of being objects are called abstract and interfaces skirt around the problem single-parent inheritance by defining empty method headers. This is the way Java, C#, and other languages solve the inheritance problem.

(example 14)
JavaScript does things radically different: It implements prototype-based programming, also called prototypical, classless or instance-based programming. As stated before, in JS there are no explicit parental or child-classes or objects. Instead, they are linked through a property: The prototype, a data construction that contains initial members and their starting values. Prototypes are accessible through the __proto__ variable, and all prototype variables are accessible to the object as if it were their own. There are two ways of creating objects: From nothing, using literals, and cloning the prototype of an existing object. However, the first way is a misnomer, as all empty objects are clones of the Root Object. The Root Object is the default prototype for all newly created empty objects. Using the object literal means cloning the Root Object prototype and placing it as the prototype of the object. As such, we are already inheriting fields, their values, and methods from an object, right from the get-go. The Root Object is a prototype for the rest of the objects. In order to inherit the properties of a previously declared and initialized object, the syntax to use is var child = Object.create(parent). What this does is that it copies the parent object and assigns it to the prototype of the child. In this manner, the child object has a direct, one to one relation with the parent akin to classical inheritance. This is the same as declaring child.__proto__ = parent, however the previous syntax results into a much cleaner solution

that does two things at once: It creates an empty object and injects it with a reference to a prototype, which is the parent. This makes two questions arise: One, what happens when we have two variables on the same object, one declared inside the object, the other inside it's prototype? What gets called? And two, what happens if we change the value within a prototype? The answer to question number one is simple: When an object has the same name for two variables inside the object body and the prototype, the most local scope takes precedence. The prototype value does not disappear, however it simply does not return if the object's fields are queried. The answer to number two is a bit trickier: If object B has by prototype object A, making A parent of B, should any change happen to the prototype of B will be reflect on A itself. Also, if after establishing the prototype relationship A is modified, those changes will be reflected on B.

This means its possible for a child to change the state of a parent. Up to now I have been using the terms child and parent, but only doing so as a crutch to transition the thought process from Java to JavaScript. The greatest strength of prototypical inheritance is that it eschews the concept of a top-down tree of inheriting data over multiple classes, or objects. Even after lending their data after establishing inheritance, the objects remain free to be modified and accessed, and their "children" will have those changes reflected on them.

However, as with everything there are multiple ways of tackling the same problem and JavaScript is no exception. Extending objects through the use of Object.create() is not the only way to create inheritance. Due to the fact that functions are also objects, they can also be extended. These are constructor functions, a holdover from classical inheritance. It's important to note that there is no syntactic difference between regular functions and constructor functions. Constructor functions can be called and assigned as fields just like regular function. However, they become constructor when they are used like so:

```
/*
	var Vehicle = function Vehicle() {
		// ...
	}
	var mazda = new Vehicle();
*/
```

The use of the new keyword may confuse you, as this is usually associated with classes and objects. The instantiation of a variable with a "new" keyword does the following: It creates a new empty object, like instantiating with the object literal. It sets the sets the object to delegate all shared data and methods to the prototype of the Vehicle() function, which means that the new object shares data with the function object, and sets the constructor property of the object to Vehicle. This is a new property that we had not discussed: In addition to being able to share data with an object through prototype, objects have a constructor property that contains a function that is executed when the object is generated by the Virtual Machine. In our previous example, whatever is contained within the Vehicle constructor is executed as many times as we create objects using /* new Vehicle() */, within the local context of the object. The constructor exists within the prototype, as object.prototype.constructor. With this, the new objects inherit all properties and functions detailed within the constructor execution. It's important to note that creating objects through Object.create() means leaving an empty constructor. This approach is the most popular since, for a time, it was the only approach for inheritance in JavaScript. However, in recent years this method has been decried as a messy way to create objects that inherit data and methods from others. That being said, this is an approach that can be seen in many frameworks and external libraries, and a JS programmer would do well to learn it. For further information, please check out David Walsh's JavaScript Objects Deconstruction.

In both the Object.create() method and the constructor/new method of inheritance, we are chaining the prototype of our parental object to that of the new objects we create with data on them. This is to delegate execution of code not found within the local scope objects and instead doing it in the objects that are already within memory. This practice removes the need for redundant code and encapsulates local variables and methods where they should be.

Another strength of JavaScript's approach to object-orientation is Object Augmentation. If we want an object that is slightly different than another, instead of writing an entire new class in Java, we can simply override or add new methods and variables to an object once it has a prototype. We can then have other objects derive from it, and so on. We can even override the native Object methods, such as .toString(), as if we were just putting something new into an object: (example 15).

If we create an object using myParenizor as a prototype, then that object has the overridden toString() method.

Another approach to inheritance, one that eschews the IS-A relationship entirely, is called Parasitic Inheritance, where a function takes another function, or functions, and incorporates its own methods and values into it and returns the modified function as if it was its own output. This pattern is also demonstrated by Douglas Crockford. (example 16)

Again, taking advantage of functions being objects in JavaScript, this approach makes it possible to cherry-pick methods and fields from various objects, put them all into one and return it, the "parasites" already ready to be used. In the words of Crockford: Parasitic inheritance is about the was-a-but-now-is-a relationship. The constructor has a larger role in the construction of the object.

However, this is not the end of inheritance in JavaScript. In the coming years, the 6th edition of the ECMAScript standard will be implemented in browsers, devices and everywhere where JavaScript is being executed. The standard, codenamed Harmony, is being partially implemented in many compilers, engines and browsers today. This year, 2014, its feature list was frozen, and it will see a full release by mid-2015. Among these features is the "Class", a data structure that is meant to emulate the classical inheritance model of Java. Under the ES6 standard, Classes will have a place in the JavaScript syntax, alongside the prototypical inheritance the language was built upon. It will look like this: (example 17)

It's important to note that the Class structure is merely syntactic sugar the previously discussed Objects and prototypes, offering a cleaner way to create objects and deal with inheritance. You can create new objects from classes and extend them in the traditional one-to-one relation. No implementation of abstracts or interfaces is planned. It was created in order to port libraries and software in a much easier way, and to facilitate adoption of a much larger developer community. For now, there is no browser that implements this structure, and the standard hasn't fully been formed yet, but it's only a matter of time until it comes to browsers and devices.

If you cannot wait for 2015, Typescript is a strong alternative. Created by Andres Hejlsberg, curated and maintained by Microsoft, Typescript is a free and open source programming language that mimics JavaScript syntax. However, it incorporates many features that bring it closer to the C-family languages: It supports the use of classes and interfaces, header files, variable type checking, and supports generics programming. Any valid JavaScript code is valid Typescript code, and it fully supports ECMAScript version 3, with support for versions 5 and 6 coming soon. Its compiler trans-codes Typescript into JavaScript, making it executable on any engine. It's fully integrated with Visual

Studio 2013, and is supported by JetBrains Webstorm. The compiler is also available as a module in npm.

For further information about JavaScript Inheritance please check out Source Decoded - Prototypes, Prototypical Inheritance done right.

ADVANCED JAVASCRIPT USES

As programs get bigger, many of the usual structures and flow control schemes get harder to understand and comb through, specially when you're jumping into a project mid-way. And as something gets more complex, the possibility for it to fail gets larger and larger over time. This is why it's important to always look for ways to keep the complexity of your code under control, which means as low as possible. Abstracting and creating inheritance chains is important, however it's easy to fixate on small issues than arise from such a practice.

Functional programming is a quick way to take care of complex code by abstracting what is trivial into less than half of the lines of code. Knowing what the fundamental functions do in every JavaScript object creates effective and cleaner code. The best example of JavaScript Functional Programming lies in the Array methods.

Traversing the elements of an array can be done in a trivial for-loop like so:

```
/**
	var names = ["Ben", "Jafar", "Matt", "Priya", "Brian"],
		counter;

	for(counter = 0; counter < names.length; counter++) {
		console.log(names[counter]);
	}
**/
```

But this can be reduced in complexity, by using the built-in method "forEach"

```
/**
	var names = ["Ben", "Jafar", "Matt", "Priya", "Brian"];

	names.forEach(function(name) {
		console.log(name);
	});
**/
```

.forEach() abstracts away the manner in which the array is traversed, and only receives an instruction to do something with each element, in this case log them into the console. This use may be petty, but it makes for cleaner code that everyone can understand at a glance.

.map() is another infinitely useful function. It acts exactly the same as forEach(), but in addition of applying the function to each element of the array, it creates a new array with the modified elements.

```
/**
	var names = ["Ben", "Jafar", "Matt", "Priya", "Brian"];
	var NAMES = names.map(function(name) {
		return name.toUpperCase();
	});
	console.log(names);
	console.log(NAMES);
}
**/
```

Again, the array traversal is hidden to privilege what it is the developer going to do with each element as it is pushed into the new array.

Another function that abstracts away code even further is the filter() function, which fills a new array

with elements of another one that return true after a comparison has been made in the callback. It "filters" your data through a condition and then returns the results that did comply with the filter.

```
/**
	var names = ["Ben", "Jafar", "Matt", "Priya", "Brian"];
	var namesB = names.filter(function(name) {
		return name.indexOf("B") >= 0;
	});
	console.log(names);
	console.log(namesB);
}
**/
```

Chaining these methods is possible to create powerful data traversal algorithms:

```
/**
	var names = ["Ben", "Jafar", "Matt", "Priya", "Brian"];
	var namesB = names.filter(function(name) {
		return name.indexOf("B") >= 0;
	}).map(function(nam){
		return nam.toUpperCase();
	});
	console.log(names);
	console.log(namesB);
}
**/
```

The .reduce() array tackles a different problem altogether. Remember the adding up all the numbers of an Array? How could we eschew the use of a traditional for loop for that matter? Reduce takes the elements of a function, an accumulator value, and lets the user decide what to do with each incoming element. The traditional way of demonstrating .reduce() is by the aforementioned arithmetics.

```
/**
	var res = [1,2,3].reduce(function(accumulatedValue, currentValue) {
		return accumulatedValue + currentValue;
	});
	console.log(res);
**/
```

The acumulator gets reset every time the function returns a value, overriding whatever was before it. The identifier is the final result. Of course this doesn't only just happen in a numerical standpoint, .reduce() can also be used for other purposes. Simply put, .reduce() traverses an array and continuously updates a variable that would normally be outside of a traditional for loop. The second argument in the reduce function is the initial value of the accumulator.

```
/**
	var names = ["Ben", "Jafar", "Matt", "Priya", "Brian"];
	var biggest = names.reduce(function(cumulator, current){
		if(cumulator < current)		return current;
		else return cumulator;
	}, names[0]);
	console.log(biggest);
**/
```

Combining all these three very distinct functions built into the Array constructor creates simpler, cleaner and in many cases faster code that does not buckle under pressure. This is simply scratching the surface of functional programming, however it servers as an appetizer to a full course of advanced

JavaScript Applications.

Promises constitute another useful feature to have in the JavaScript arsenal, seeing as much of the initial use of the language was to capture user input and then creating changes in the View. Promises are not a part of the ECMAScript standard, however they are a coming feature in Harmony and external libraries such as JQuery and q support it. In short, promises are methods that return a dynamically generated callback once the execution of a certain function has been completed. Remember the old adage in computer programming that you cannot be 100% sure a function has completed and delivered data in time for you to start another function with that data? Promises fix that by ensuring a callback is executed right after the function that makes the promises finishes its own execution. Promises are either resolved, where the function was completed successfully, or rejected, where there was an error in computation and the function failed to deliver that promise. The code in ES6 will look like this, with similar implementations coming in q and jQuery today.

```
/**
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then…

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
**/
```

Modules are at the heart of loosely-coupled code, much like packages and namespaces in other languages. Modules are discreet, self-stored packages of functionality that only expose their public methods when they are necessary. Implemented efficiently, Modules can bring pre-packaged code that can maximize the potential of a codebase. The current iteration of JavaScript doesn't support modules natively, it will do so in ES6, however two standards have sprung up to solve this problem and create modular applications: AMD, or Asynchronous Module Definition, and the CommonJS standard. The specifics of defining this module and how do they work is beyond the scope of this seminar, however we can briefly discuss their strengths and weaknesses. AMD is more suited for client-side scripting, as it loads code asynchronously using native JavaScript code. It's highly flexible, and currently embraced by larger projects such as jQuery and Firebug. The CommonJS standard, on the other hand, sprung from volunteers who decided to design and standardize the JavaScript APIs, so much fragmented by the browser wars the language was subjected to. The CommonJS structure is simpler than that of AMD, and a lot more straight forward to understand, however it is not supported in any browser or framework beyond Node.js. EcmaScript 6 adopts neither module schematic, instead adopting its own syntax with two new keywords: import and export. For further information on modules, please check out Addy Osmani's Writing Modular JavaScript with AMD, CommonJS and ES Harmony.

We have barely scratched the surface of Advanced Uses of JavaScript, but covering them all in depth would take hours upon hours of seminar. For more information I urge you to checkout nodeschool.io, they provide CLI learning tools for you to pick up and start experimenting with advanced JS features for free. They are served on npm, so it's necessary to get Node.js for your system before starting to play with them.

EXTERNAL LIBRARIES

JavaScript is not only limited to syntax and inheritance models. Its simplicity is deceiving, but the programming language has garnered thousands of supporters and active contributors, creating a community that has taken the language to its limits and beyond. Used right, this little oddity, birthed out of the savage competition of who would control content on the internet, has come a long way and nowadays it commands respect among older and more established languages in terms of portability, flexibility and a chameleon ability to take on the form users need. If it doesn't have an implementation in JavaScript, rest assure someone in the community is hard at work writing a library that is most likely Open Sourced and under the GNU, BSD, or Creative Commons licenses. The greatest strength of JavaScript, more than its features, is its vibrant community that accomplishes feats of code that months ago were thought of as difficult or even impossible.

The following will be just a rundown of Third-Party, External JavaScript Libraries all available for download at absolutely no cost.

jQuery is perhaps the most popular JavaScript Library of them all. Its omnipresence so big almost to the point of annoyance for purists of the language, John Resig's work has shaped the course of the language, its user-base, and ease of access. jQuery is, first and foremost, a library to modify the Document Object Model. CSS Animations, Mouse Clicks, Events, Keyboard Clicks and Markup Manipulations are jQuery's specialty. JavaScript has a DOM API that is known to be cumbersome at best, counter-intuitive at worst. Not every web browser implements changes to markup the same, not every web browser renders HTML and CSS the same way, and no browser executes JavaScript in the same manner. jQuery is a library that is prepared to account for those changes and put absolute control in the hands of developers, without nasty surprises or caveats that make them write client-specific code. While it's true the use of jQuery may make some developers lazy and dependent of the tool, it's effectiveness is never questioned. Among jQuery's products there's a UI library, a mobile-specific version, and a unit testing framework. Most visual frameworks, like Zurb Foundation, Twitter Bootstrap, and Sencha Touch, are built on top of jQuery. There are tens of thousands of plugins that extend jQuery's functionality, and growing.

The Phaser Game Engine is a production-tested, well-maintained, and comprehensive open source game framework to make desktop and mobile browser HTML5, leveraging the power of the HTML5 Canvas element and WebGL. Developed by Richard Davey of Flixel Power Tools fame, it supports multiple kinds of sprite sheets, sprite grouping, particles, multiple cameras, touch and keyboard inputs, has multiple sound channels, renders tile maps, has a plugin system and it's been widely accepted as one of the best game engines available for the browser. It has been deployed for games featured in BBC and Discovery Kids, has powered a great percentage of Ludum Dare entries and has an ardent community of supporters and developers that write plugins and tutorials constantly and consistently. The engine is updated constantly with bug fixes, improving features and maximizing performance. The engine supports both native JavaScript and Typescript, and its available for free in GitHub.

If 3D Games are more your cup of tea, Three.js offers a widely available and free option. Created by Ricard Cabello, Three.js is a library used to create and display animated 3D computer graphics within a webpage, using the OpenGL subset specifically for browsers: WebGL. Three.js abstracts away shader and material code to offer an easier way to implement models, textures, and lighting. It supports Canvas rendering, orthographic and perspective cameras, forward and inverse kinematics, lambert and phong materials, GLSL shaders, data loaders, supports JSON data ported from Blender, and is bundled with a Debugging Inspector. Experiments and full fledged complete projects are available on the

website threejs.org.

If graphics are not of your interest, building applications on the web has become a cinch in recent years with MV* libraries. Following the time-tested practice of separating Data Storing, Data Handling, and Communication with the User into the MVC paradigm, a series of libraries have made the work of web developers easier by taking care of boilerplate code, browser, and language quirks. The * in MV* accounts the variety of solutions people have come up with in regards of figuring out how to make Model and View communicate with each other. There are many MV* frameworks, but Angular.js and Backbone.js are the most used and with good reason. Angular, developed and maintained by Google, assists the creation of single-page applications, simplifying its development and testing. The library makes use of custom HTML tags and attributes, directives to tell Angular to bind input and output to certain parts of the markup. It decouples the markup from application logic, and provides a loosely coupled structure. Backbone.js is another library that follows the MV-Presenter paradigm, and my personal tool of choice. Backbone is a lightweight, general purpose library that is designed to keep Model and View synchronized with each other, sending and receiving events that prompt changes within the markup. It's lightweight, un-opinionated and requires templates written in underscore.js rather than special tags and attributes in the markup. It's also quite extensible, with a wealth of plugins that complement Backbone's functionality.

If building applications for the web is not of your interests, how about building applications for all mobile operating systems at once? Apache Cordova, and it's closed-source implementation Adobe Phonegap, is a mobile development framework that enables developers to build applications for mobile devices using JavaScript, HTML5 and CSS. Instead of relying on separate native-code APIs for each platform, Cordova enables users to create one codebase for all platforms. It supports deployment to iOS, Symbian, Android, FirefoxOS, Ubuntu Touch, webOS, Windows Phone and Windows 8. The core of PhoneGap applications use HTML5 and CSS3 for their rendering, and JavaScript for their logic. Although HTML5 now provides access to underlying hardware such as the accelerometer, camera and GPS, browser support for HTML5-based device access is not consistent across mobile browsers, particularly older versions of Android. To overcome these limitations, the PhoneGap framework embeds HTML5 code inside a native WebView on the device, using a foreign function interface to access the native resources of the device. Native plug-ins allow developers to add functionality that can be called from JS, allowing for direct communication between layer and JS logic. This enable support for the accelerometer, camera, microphone, compass and file system. However this comes with the caveat that apps created on Phonegap are slower than their native counterparts, and may be rejected by Apple.

Node.js is the prime example of the power JavaScript has, and what ultimately me sold me into the programming language. Despite it has only been around since 2009, it has taken the world of networking applications by storm. Node.js, developed by Ryan Dahl and maintained by Joyent, is an open-source, cross-platform runtime environment for server-side applications. In short, it's server code written in JavaScript. It uses the Google V8 engine to execute code, with an extensive library of plugins and modules in its own package manager: npm. Since the adoption of this package manager in 2011, Node.js has seen an explosion of support from both small-time developers to large corporation such as Walmart and Paypal. Mozilla has based its own two-step authentication software, Mozilla Persona, on Node.js. It enforces non-blocking asynchronous I/O execution based around a single-threaded event loop. This is possible through the implementation of function callbacks, another benefit of having functions as objects. Circumventing in this manner its weakness of being a single-threaded process, Node.js can keep many connections alive without having to return a "busy server" status. Combining Node with a document DB and JSON serialization offers a unified development stack that

allows what were traditionally client-side paradigms, such as MVC, MVP, and MVVM, Node.js allows the reuse of models and services, with the added bonus of having an entire application written in a single language. Its extensibility through the package manager has allowed Node to break out of the server and enter other technologies successfully: Socket.io is a library that establishes a two-way communication pipeline from client to server. It's based on the WebSocket connection protocol, and enables response times with under 200 ms latency. Multi-player games and chatrooms can be written using Socket.io to guarantee quick, swift response times. Nodeuino, on the other hand, is a framework that uses Socket.io and Node.js to access Arduino boards. It enables a JavaScript interface with the Arduino, enabling the user to make changes to the board loop in real time, all the while listening to events generated by Button pushes. The library is mostly in an alpha stage, but it's already showing great promise. If none of these applications sound like something you'd like to do, and you'd rather build strictly computer software, node has an answer for you through the node-webkit library. Node-webkit is an app runtime based on Chromium and node.js, enabling developers to write native apps in HTML and JS. Node-webkit wraps the developer's code in a Chromium Embedded Framework, basically a single-tab, stripped down version of chrome, with the V8 engine built in. This has the advantage to be platform agnostic, and runs on all major Operating System. It makes applications easy to package and distribute. Again, it's a nascent technology, but it is promising.