Clearing the Air to Javascript

A beginners reference

Ryan Morris
4-22-2016

Table of Contents

Forword

This guide should be used in conjunction with other Javascript material and is not intended as a standalone resource to learn Javascript. Everything in this guide is focused on what I have been exposed to as a beginner and is not a complete or total representation of the Javascript (JS) language. Hopefully, this guide can bring some clarity to some of you and act as a refresher for others.

The basics cover much of what I learned from courses on Udemy.com to codecademy.com and the various forums and posts online, all of which will be linked to at the end. I found that on average I understood around 80% -85% of the material and had to conduct my own research and start my own projects to bring it all together. I also had trouble retaining information in the beginning. I found that by focusing on one area at a time and then practicing, actually coding, I retained a lot more. there are tons of sites that offer code challenges. I used this as a reference to help me organize my lessons, and as my personal notes, and I thought it would be fun to try and make it into a guide. As I said above, this does not include everything, but it is a good resource for beginners.

Javascript: Under the Hood

Here, I will write a bit about what happens when Javascript (JS) runs code. This is important to understand because you need to know how Javascript takes your code and makes sense of it then relays your code to the computer in a way it can understand what you want to accomplish.

How JS Reads Code

This area will include basic information for how JS reads your code. First off, JS is single threaded and synchronous. Essentially, this means that JS will execute one code and one line of code at a time and in order. JS does have some features that allow it to be asynchronous (more than one at a time) but that is for a later topic.

Syntax Parser

A program that reads code and determines what it does and if it is valid. A syntax parser passes your code through certain steps, validates it then sends it along to the computer in a way the computer can understand.

How does this work? Your code ➔ compiler/interpreter ➔ computer

Execution Context

A wrapper that helps manage the code you write. It can contain other things beyond the code you have written such as JS's built in functions. There are two phases: the creation and execution phases. The creation phase sets up the environment (more on environments below), while the execution phase runs your code.

The Execution Stack

Every function that is invoked creates a new execution context which is added to the execution stack. Code that is currently running is placed on the top of the stack. When a piece of code finishes running it is removed or "popped" off of the stack. When you first run your code there is a **Global Execution Context** that is created during the "creation phase" and that stores functions and variables in memory.

Hoisting

Memory is set aside for functions and variables when your code is ran. When your code is executed, JS will look for your functions and variables in this space. Functions are saved in their entirety, however, "undefined" is used as a placeholder for variables until they are assigned a value. This is why you can call a function before that function but you cannot return the value you set for a variable before that variable is assigned that value. If you tried that, JS will return "undefined."

Variable Hoisting

It is important to note that within a function all variable declarations will be hoisted to the top of the function. A declaration is simply when you declare a variable i.e.: var foo; This does not happen when you define or assign a value to a variable i.e.: var foo = 0;. However, function declarations take precedence over variable declarations and will be hoisted above them and ran first.

Ex. var food;

```
function food() {}          //if you console.log food you will get the function not the variable
```

Function expressions are not hoisted. See more about functions below.

Javascript: Environments and Scope

Environments

Global Environment

Global object "window" is inside every browser and is part of the "Execution Context." In this environment you will find built in functions. You can access it with dot notation.

Ex. syntax: window.*"function";*

Variable Environment

This is where variables are and are stored as to how they relate to each other in memory. Even if a variable is declared multiple times with different values the variables will remain distinct and will not overwrite each other, however, the last declared value will be placed in the **Global Execution Context** and that will be the value returned.

Ex. var shout = "yell";
var shout = "scream"; *//JS will return the "scream" but there are essentially*
// two "shout" variables with different values

Dynamic Typing

The JS engine determines what type of data a variable holds while a code is running. This is why variables can hold different types of data.

Ex. var type = "hello"; *//You do not need to tell Javascipt this is a string*
(inside function) var type = true; *//2 separate var types now hold a boolean and string value*

Scope

Lexical Environment

Where something sits within the code you write. Where you write something is important. For example creating variables inside of functions and global v. local scope.

Local and Global Scope

If a variable is declared outside of a function with the var identifier then it has global scope and can be used by other blocks of code. If it is declared within a function the variable has a local scope and is terminated at the end of that function and cannot be used outside of the function. You can, however, declare global variables inside of functions. Just don't use the reserve word "var."

```
Ex. var ryan = "Ryan";          // ryan now has a global value of "Ryan"
 function ryanName (ryan); {
    var ryan = "Ryan Morris";    //Now ryan ="Ryan Morris" but only when you call the function
};
```

Note: you must use the identifier var to declare local variables in a function.

```
Ex. function ryanName(ryan); {
      name = "Ryan Morris";       // If there is no other variable "name" this line just created a global variable
                                  //In other words, "var" is a scoping identifier
};
```

Remember, if you declare a global variable inside of a function it can be used outside of the function. However, if that variable was already defined then it will change that variable so be careful.

Ex. var fast = "Fast";

```
function cars(car1, car2) {
        fast = "100 mph";        //After "cars" is ran fast will be equal to "100 mph" and not "Fast"
};
```

Scope Chain

Global vs. Local scope. i.e.: When a function is inside another function it is not added to the Global Environment and that is why you cannot call a function that is inside another function until the parent function is ran. Also, checks if a variable is a new one or a copy i.e.: same variable with different values saved in memory.

Key Words, Operators, Precedence and Comparisons… Oh My!

Statement Identifiers

Also called reserved words. **They cannot be used as variable or function names**. Reserved words are built into JS and have specific functions that cannot be changed.

Statement & Description

break	Exits a switch or a loop
continue	Breaks one iteration (in the loop) if a specified condition occurs, and continues with the next iteration in the loop
debugger	Stops the execution of JavaScript, and calls (if available) the debugging function
do ... while	Executes a block of statements and repeats the block while a condition is true
for	Marks a block of statements to be executed as long as a condition is true
for ... in	Marks a block of statements to be executed for each element of an object (or array)
function	Declares a function
if /else/else if	Marks a block of statements to be executed depending on a condition
return	Stops the execution of a function and returns a value from that function
switch	Marks a block of statements to be executed depending on different cases
throw	Throws (generates) an error
try/catch/finally	Marks the block of statements to be executed when an error occurs in a try block, and implements error handling
var	Declares a variable
while	Marks a block of statements to be executed while a condition is true

Note: There are more. These are just the basics.

List of reserved words

abstract arguments boolean break byte	case catch char class* const	continue debugger default delete do
double else enum* eval export*	extends* false final finally float	for function goto if implements
import* in instanceof int interface	let long native new null	package private protected public return

short static super* switch synchronized	this throw throws transient true	try typeof var void

List of Operators

In Javascript operators are used to assign and compare value, perform arithmetic operations and more. They are a special function and, generally, take two parameters and return one result. There are three ways to use them:

- Infix notation: function name is between two parameters //3+3
- Prefix/postfix: function is before/after parameters respectively // --3, 3++

I found the following list at: http://web.eecs.umich.edu/~bartlett/jsops.html

Operator Category	Operator	Description
Arithmetic Operators	+	(Addition) Adds 2 numbers.
	++	(Increment) Adds one to a variable representing a number (returning either the new or old value of the variable)
	-	(Unary negation, subtraction) As a unary operator, negates the value of its argument. As a binary operator, subtracts 2 numbers.
	--	(Decrement) Subtracts one from a variable representing a number (returning either the new or old value of the variable)
	*	(Multiplication) Multiplies 2 numbers.
	/	(Division) Divides 2 numbers.
	%	(Modulus) Computes the integer remainder of dividing 2 numbers.
Comparison Operators	==	Returns true if the operands are equal.
	!=	Returns true if the operands are not equal.
	>	Returns true if left operand is greater than right operand.
	>=	Returns true if left operand is greater than or equal to right operand.
	<	

		Returns true if left operand is less than right operand.
	<=	Returns true if left operand is less than or equal to right operand.
Logical Operators	&&	(Logical AND) Returns true if both logical operands are true. Otherwise, returns false.
	||	(Logical OR) Returns true if either logical expression is true. If both are false, returns false.
	!	(Logical negation) If its single operand is true, returns false; otherwise, returns true.
Bitwise Operators	&	(Bitwise AND) Returns a one in each bit position if bits of both operands are ones.
	^	(Bitwise XOR) Returns a one in a bit position if bits of one but not both operands are one.
	|	(Bitwise OR) Returns a one in a bit if bits of either operand is one.
	~	(Bitwise NOT) Flips the bits of its operand.
	<<	(Left shift) Shifts its first operand in binary representation the number of bits to the left specified in the second operand, shifting in zeros from the right.
	>>	(Sign-propagating right shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off.
	>>>	(Zero-fill right shift) Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off, and shifting in zeros from the left.
Assignment Operators	=	Assigns the value of the second operand to the first operand.

	+=	Adds 2 numbers and assigns the result to the first.
	-=	Subtracts 2 numbers and assigns the result to the first.
	*=	Multiplies 2 numbers and assigns the result to the first.
	/=	Divides 2 numbers and assigns the result to the first.
	%=	Computes the modulus of 2 numbers and assigns the result to the first.
	&=	Performs a bitwise AND and assigns the result to the first operand.
	^=	Performs a bitwise XOR and assigns the result to the first operand.
	|=	Performs a bitwise OR and assigns the result to the first operand.
	<<=	Performs a left shift and assigns the result to the first operand.
	>>=	Performs a sign-propagating right shift and assigns the result to the first operand.
	>>>=	Performs a zero-fill right shift and assigns the result to the first operand.
String Operators	+	(String addition) Concatenates 2 strings.
	+=	Concatenates 2 strings and assigns the result to the first operand.
Special Operators	?:	Lets you perform a simple "if...then...else"
	,	Evaluates two expressions and returns the result of the second expression.
	delete	Lets you delete an object property or an element at a specified index in an array.
	new	Lets you create an instance of a user-defined object type or of one of the built-in object types.
	this	

		Keyword that you can use to refer to the current object.
	typeof	Returns a string indicating the type of the unevaluated operand.
	void	The void operator specifies an expression to be evaluated without returning a value.

Order of Precedence and Comparison

For arithmetic code JS reads the code just as you would a math problem, PEMDAS: Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction. It is important to remember **Operator Associativity** or how JS will read your code i.e.: Left-to-Right or Right-to-Left.

Here are two very important references for operator associativity and comparisons. Both of these were taken from Tony Alicea's course on Udemy.com, "Javascript: Understanding the Weird Parts," which I highly recommend. To be fair, he did get them from another site which is linked below each table.

Highest precedence will run first and has the highest number.

Precedence	Operator type	Associativity	Individual operators
19	Grouping	n/a	`( … )`
18	Member Access	left-to-right	`… . …`
	Computed Member Access	left-to-right	`… [ … ]`
	new (with argument list)	n/a	`new … ( … )`
17	Function Call	left-to-right	`… ( … )`
	new (without argument list)	right-to-left	`new …`
16	Postfix Increment	n/a	`… ++`
	Postfix Decrement	n/a	`… --`
15	Logical NOT	right-to-left	`! …`
	Bitwise NOT	right-to-left	`~ …`
	Unary Plus	right-to-left	`+ …`
	Unary Negation	right-to-left	`- …`
	Prefix Increment	right-to-left	`++ …`
	Prefix Decrement	right-to-left	`-- …`
	typeof	right-to-left	`typeof …`
	void	right-to-left	`void …`
	delete	right-to-left	`delete …`
14	Multiplication	left-to-right	`… * …`
	Division	left-to-right	`… / …`
	Remainder	left-to-right	`… % …`
13	Addition	left-to-right	`… + …`

	Subtraction	left-to-right	… - …
12	Bitwise Left Shift	left-to-right	… << …
	Bitwise Right Shift	left-to-right	… >> …
	Bitwise Unsigned Right Shift	left-to-right	… >>> …
11	Less Than	left-to-right	… < …
	Less Than Or Equal	left-to-right	… <= …
	Greater Than	left-to-right	… > …
	Greater Than Or Equal	left-to-right	… >= …
	in	left-to-right	… in …
	instanceof	left-to-right	… instanceof …
10	Equality	left-to-right	… == …
	Inequality	left-to-right	… != …
	Strict Equality	left-to-right	… === …
	Strict Inequality	left-to-right	… !== …
9	Bitwise AND	left-to-right	… & …
8	Bitwise XOR	left-to-right	… ^ …
7	Bitwise OR	left-to-right	… | …
6	Logical AND	left-to-right	… && …
5	Logical OR	left-to-right	… || …
4	Conditional	right-to-left	… ? … : …
3	Assignment	right-to-left	… = …
			… += …
			… -= …
			… *= …
			… /= …
			… %= …
			… <<= …
			… >>= …
			… >>>= …
			… &= …
			… ^= …
			… |= …
2	yield	right-to-left	yield …
1	Spread	n/a	... …
0	Comma / Sequence	left-to-right	… , …

Javascript – Equality Comparison and Sameness

x	y	==	===	Object.is

undefined	undefined	true	true	true
null	null	true	true	true
true	true	true	true	true
false	false	true	true	true
"foo"	"foo"	true	true	true
{ foo: "bar" }	x	true	true	true
0	0	true	true	true
+0	-0	true	true	false
0	false	true	false	false
""	false	true	false	false
""	0	true	false	false
"0"	0	true	false	false
"17"	17	true	false	false
[1,2]	"1,2"	true	false	false
new String("foo")	"foo"	true	false	false
null	undefined	true	false	false
null	false	false	false	false
undefined	false	false	false	false
{ foo: "bar" }	{ foo: "bar" }	false	false	false
new String("foo")	new String("foo")	false	false	false
0	null	false	false	false
0	NaN	false	false	false
"foo"	NaN	false	false	false
NaN	NaN	false	false	true

Basics: Code, Best Practices and Stuff

camelCase

Best practice for how you type in JS. Simply capitalize the first letter of every word after the first word.

Ex. contactList, frenchFries

Whitespace

Invisible characters that create space in your code i.e.: enter, spacebar and tab. The sole purpose of whitespace is to make your code more readable. JS will ignore whitespace.

Automatic Semicolon Insertion

JS will insert a semicolon automatically in certain circumstances. For example, if you are trying to "return" and hit "enter" then put your code, JS will insert a semicolon after return and not run your code

```
Ex. function test() {
        return                          //JS will not return "hi" because when you hit enter it inserted a
                                        //semi-colon after return which would return undefined
        "hi";

}
```

Primitive Data, Default Values and Coercion

A type of data that represents a single value. I.e.: not an object (object is a collection of pairs) is known as a primitive data type. Javascript has **5 primitive data types**: Numbers, Booleans, Strings, Null, Undefined.

Default Values

Javascript will use undefined for variables/parameters that are not defined with a value. Don't set variables to undefined. Undefined means that a variable does not have a value. There are other types of default data in regards to the global and other objects. Those will be elaborated on in their relevant sections. Just be aware that there are default values assigned to variables and properties of objects.

Coercion/Type Casting

Converting a value form one type to another is called coercion when done implicitly and type casting when done explicitly. What does that mean? It's probably best if we use examples here:

```
Ex 1. var foo = true;
        if(foo == 1)        //This is coercion because the " == " will convert the boolean to true (1)
                            //The conversion is done in an unobvious manner
        1 + "2" = "12       //Also coercion

Ex2. var bar = 100;
        toString(bar);      //This will convert var bar from a number 100 to a string "100". This is obvious
                            //and explicitly stated so this is type casting
```

By Value v. By Reference

When you assign values or create objects it is important to keep two things in mind:

- By Value (primitive types): when you set a variable equal to another's value you "copy" the value.
- By Reference (objects including functions): when you set a variable equal to an object you point to the location of the object. *//See Inheritance and Prototype below*

Ex. var a = 1;
var b = a; *//b now equals 1. This is by value*

Variables

Variables are used to store information. Use the reserved word "var" to declare a variable, you don't always have to but it is good practice. Variables are case sensitive and should start with a letter.

Syntax: var variableIdentifier = variableValue;

Ex. var ryan = "Ryan*";* *//Stores information/value "Ryan" to var ryan*

To change a var do not use var, use the variableIdentifier

Ex. ryan = "Ryan loves Aileen";

Now var ryan's value has been changed.

Functions

A function takes in an input and produces an output based on whatever tasks (functions) you assigned it. There are many different ways for writing a function you can right a function declaration, an anonymous function and define a function. **Functions are objects, except functions can have invocable code and be called.**
Function Statements and Expressions

- Expression a unit of code that results in a value
  - Can create function expressions by wrapping the function inside parenthesis *(function)*
  - IIFE stands for Immediately Invoked Function Expressions. This is when you wrap a function in parenthesis and then call it
    - Ex. (function test() {*Does something*})(); *//I don't really know why you would use this*
- Statement a unit of code that does work but does not return a value
  - Function's are statements until called
    - Unless you use anonymous functions (no name) such as with variables
      - Ex. var funFunction = function () { console.log("yo"); }

Arguments || Parameters

These are simply what you pass to a function between the closed parentheses and are the same thing in regards to functions. As a keyword it contains a list of all the values of the parameters you pass to a function. Similar to an array but it is not an array, however, you can use functions such as .length or bracket notation on it.

Ex. 1: if(arguments.length > 0) {}
Ex. 2: arguments[0]; *//Bracket notation*

Defining a Function

You use the reserved word "var" to define a function.

Syntax: var functionName = function(property) {
//do something;

```
};                          //The semicolon isn't needed but I was told it is best practice to use
```

Function Declaration

A function declaration is simply creating a named function without using the reserved word "var" to assign the function to a name.

```
Syntax: function name (property) {
            //do something
        }
```

Anonymous Functions

This type of function is mostly used in closures and as arguments for other functions. When you create an anonymous function it cannot be initialized like a declared or defined function.

```
Syntax: function() {
            //do something
        }
```

Calling a function

You use functionName(argument) to call a function. The () invoke the function so it is not necessary to have a function name. You can use an IIFE as well.

```
Syntax: functionName(property);
Ex. greetings("Ryan");
```

Return

You can use the keyword "return" to return a value from a function. Return simply outputs the value from the function. If you try to use a function without returning a value from it you will get undefined.

```
Ex. var mathMagic = function (firstNum, secondNum) {
            return firstNum * secondNum;
};

mathMagic(1, 2);            //Returns 2. That's it.
```

Higher Order Functions: Closures and Callbacks

Higher order functions **are simply functions that accept other functions as arguments**. While it sounds simple enough it can be pretty confusing to get started. Here are some examples of the higher order functions I have used the most:

- forEach

```
    syntax: object.forEach(function(itemName) {          //ForEach is similar to a for/in loop
                return do something
            });
    Ex. [1,2,3].forEach(function(itemInArray) {
                return console.log(itemInArray);         //Will simply log each 1,2,3
            });
```

Note: forEach will iterate over each item in an array just like a for/in loop.

- map

```
syntax: object.map(function(itemName) {
            return do something
        });
Ex. [-1,2,3,0].map(function(element) {      //will create a new array with the results that
        return element > 0;                 //are returned
});
```

- filter

```
syntax: object.filter(function(itemName) {
        return do something
});
Ex. [-1,2,3,0].filter(function(element) {     //will create a new array that meets the condition
        return element > 0;                   //the condition.
});
```

Note: Map and filter are similar with one key difference: **map is used to convert each item in an array** whereas **filter is used to select, or filter, each item in an array**.

- reduce

```
syntax: object.reduce(function(previousValue, currentValue, initialIndex, array){
        return do something
}, initialValue );
Ex. [1,2,3].reduce(function(a,b) {
        return a+b;                          //this will give you the sum for each item in this array
});
```

Note: There are a ton of ways to use reduce and I'd recommend you do more research about it on your own. One of the most commons ways for me so far is to flatten a multidimensional array into a single array. See more in the arrays section below.

- sort
  - syntax: object.sort(function(a,b) {
    - return a-b; //you can use just object.sort();
  - }); //if you want the ascending order
  - Ex. [2,1,3].sort(); //will return [1,2,3]
  - Ex.2. [2,1,3].sort(function(a,b){
    - return b-a; //will return [3,2,1]
  - });

Note: .sort() will sort the object based on the Unicode value of each character. Unicode is an encoding standard that assigns values to characters.

Closures

Closures are functions within other functions that have access to and retain the outside function's scope. Basically, this means that a closure function will be able to access variables and functions of the function it is inside after that function has already executed.

```
Ex. function myName(name) {
        var getName = function() {          //the internal function can access the local variable
                console.log(name);          //of the external function
        };
        getName();                          //when you run myName() it will run getName()
    }
```

Think of a button or a form on a webpage that was created using JS. Now, when that webpage is loaded the JS code that deals with that button is ran as well. But, you can add a closure to the function governing that buttons behavior such as an event i.e.: when you click the button = run the closure. This is what makes closures useful. Because the closure has access to and retains the outside function's scope and state it can use that code when it runs during the assigned event.

Callbacks and Asynchronous Behavior

There are two types of callbacks: synchronous and asynchronous. A callback function, also a closure and a higher order function, is passed to another function as a parameter and then invoked inside that function. We already discussed what synchronous means above so we won't go into it again but asynchronous, in regards to callbacks, means that the code can be executed in any order. Just know that the above higher order functions i.e.: forEach, map, etc., are synchronous callback functions.

Callbacks are a bit complicated to go into detail about. I recommend reading this page at:
http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/

It's actually one of my favorite topics elaborated on by "Javascript Is Sexy."

Loops

Loops run the same code continuously until a set condition is met. The most common loops are for, for/in, and while. Each loop serves a different function and you should use the best loop for the task.

For

The "for" loop creates a loop that holds up to three expressions. Usually used when you know the number of times the code must run.

```
Syntax: for (start; end; increment) {
            // Do something. Make sure the loop has a way to terminate or it will crash your browser
}
Ex. for (var i = 100; i >= 1; i-=5) {        //Loop starts at 100 and stops when var I is equal to 1
        console.log(i);                      //Decreases by 5 each loop and prints each value
}
Ex. of an infinite loop    for (var i = 0; i < 0; i++) {}              //Can never be less than 0
```

For/In Loops

This type of loop is usually used with array objects. You can use a for/in loop to extract properties or values from an object.

Properties from an Object

```
Syntax: for (var i in object) {              //Accesses a property(s) in an object
            //do something;
}
Ex.  for (i in friends) {
        console.log(i);                      //Prints to console "i" property(s) in the friends object
}
```

Values from an Object

```
Syntax: for(i in object) {                   //Accesses a value(s) in an object
            console.log(object[i]);
```

```
}

Ex. for(i in friends) {
        console.log(friends[i]);          // Prints to console "i" value in the friends object
}
```

While

While loops continuously loop through code until a certain condition is met, which is true or false. Usually used when you do not know how many times the code must run.

```
Syntax: while (condition) {                  //Condition is used in terms of true or false
            //do something;
            Condition terminates;            //Provide a way for the loop to terminate
}
Ex. var i = 0;

    while(i === 0) {                  //While 'i' is equal to 0 (true)
        console.log('test');          //Print test
        var i = 1;                    //Change 'i' to equal 1. Now the condition will read as false
}                                     //and the loop will terminate
```

Do/While

Uncommonly used. I have taken a few online classes in Javascript and have been told that my instructors have never even used the do/while loop so I did not feel a need to cover it here.

Arrays

Arrays are variables that can hold more than one value. The items listed in an array will start at position 0. There are two main ways to declare a new array.

```
Syntax 1: var name = new Array ():
            name[0] = "property";          //Even though this is first in the array it starts at 0
            name[1] = "property";
Syntax 2: var name = ["property0", "property1"];   //Separate properties with a " , " This is
                                                    // the easiest and most common form to use
Ex. var cars = ["Honda", "Chevy"];

cars[0];      //This calls the first item in the array "cars" which is Honda
```

Adding to an Existing Array

```
Syntax: arrayName[postion of new item] = newItem

Ex. cars[2] = "Ford";          //Now the cars array has 3 items in it: ["Honda", "Chevy", "Ford"]
```

Nested and Multi-Dimensional Arrays

Nested arrays refer to arrays that have other arrays inside of them and multi-dimensional arrays are nested arrays inside of nested arrays, essentially. Probably better if I show you.

Ex 1 Nested: var array = [[1, 2], [3, 4]];

Ex 2 Multi: var array = [1, [2, 3, [4]]];

Note: **When accessing nested or multi-dimensional arrays you need to use multiple brackets**. For example, for the nested example above: array[1][0] = 3.

Conditional Statements: If, If/Else If/ Else & Switch

You use conditional statements when you want your commands to perform certain actions based on different conditions. You can use logical operators in your conditional statements.

Logical Operators with Conditional Statments

Logical operators are && === and, || === or and ! for not. A list of operators can be found here.

If/Else

```
Syntax: if (condition) {
                    statements_1
} else {                                    //Else will run when the if condition is not met
                statements_2
}

Ex. if(x  <  100) {
        console.log("Yup  it is");          //Whenever x is less than 100 it will print this message
} else {
        console.log("No  it is not");       //Whenver x is equal to or greater than 100 this message
}                                           //will print
```

Note: using operators such as or (||), & (&&), etc., will allow you to add to your conditional statement.

If/ Else If

The addition of the "else if" conditional statement allows you to create additional conditions and actions for your code. You can use as many "else if" statements as you want but your conditional statements should start with an if statement

```
Syntax: if (condition1) {
                    statements_1
} else if (condition2){                     //Else if will run when the if condition is not met
                    statements_2
}


Ex. if(x  <  100) {
        console.log("Too  small");
} else if (x > 100) {
        console.log("Too  big");            //Whenver x is greater than 100 this message will print
} else {
        console.log("Just  right");         //Else will run only when both if/else if statements are not
}                                           //true. In this case when x is equal to 100
```

Switch

In the switch statement you **check multiple cases against an expression**. It will compare the value of the expression against the value in each case and will stop when both values match. Then the code will execute in whichever case matched the switch expression value.

```
Syntax: switch (expression) {               //Whatever you are testing against
                case propertyValue:
```

```
                    //do something
                    break;          //This breaks the statement if the preceding case is true
}

Ex. switch(x) {
                case 1:
                    if(x % 2 === 0) {console.log("X is even");}
                    break;          //If X is even then it will stop running and print the message
                case 2:
                    if(x % 3 === 0 || 1) {console.log("X is odd");}
                    break;
}
```

Note: You can use as many case statements as you need to

Objects

An object is an unordered list of certain data types such as String, Date, Array, etc., that is stored as a series of name-value pairs. Each item in the list is called a property (functions are called methods). You can add other objects to existing objects. Essentially, objects are a collection of name/value pairs and other objects.

There are a few things to keep in mind regarding Objects:

Difference between reference and actual value when storing data in objects

- Reference: when the actual data changes the property that holds the reference will change as well
- Actual: when you set a property to an actual value it will not change even if the value changes.

Objects have three attributes that are set to true as default

- Configurable: Specifies whether the property can be deleted or changed.
- Enumerable: Specifies whether the property can be returned in a for/in loop.
    - Properties that are inherited are not enumerable *//See Inheritance below*
- Writable: Specifies whether the property can be changed.

Creating Objects

Literal Notation

Most common and the easiest way.

```
Syntax: var myObject = {
                key: value,         //Known as a key-value pair and the objects properties.
                key: value,
                key: value
};

Ex 1.
 var me = {
   name: 'ryan',
   age: 29
};
```

You can add other objects using literal notation.

```
Ex 2.
var me = {
        name: 'ryan',
        address: {                          //Another object is created using { }.
                street: 'California'

        }
};
```

Note: Literal notation creates objects all at once.

Object Constructor

The second most common way to create objects is with Object constructor. A constructor is a function used for initializing new objects.

```
Syntax: var myObject = new Object();        //Creates an object with no properties so you must
                myObject.key = value;       //add using myObject.key = value;.
```

```
Ex. var myCar = new Object();
        myCar.make = "Ford";
        myCar.model = "Mustang";
        myCar.year = 1969;
```

Patterns for Creating Objects

To help streamline the process for creating multiple objects with the same prototypes, properties, methods, etc., it is best to use a pattern such as the custom constructor or the hybrid prototype constructor. There is a third pattern called the prototype constructor but it is the most tedious and everything I read recommends the hybrid pattern. **When using patterns it is best practice to capitalize the first letter of your constructor name to distinguish the function from other functions.**

Custom Constructor

A custom constructor is also called a function constructor. This type of constructor is use.

```
Syntax: var object = new function(properties) {
        key: value;
};
```

```
Ex. function Person(name,age) {         //Shorthanded version
        this.name = name;
        this.age = age;
}

var bob = new Person("Bob Smith", 30);  //Calls the custom constructor to create a new object
```

Hybrid Constructor

Similar to a custom constructor, a hybrid prototype constructor simply includes the prototypes inside of the constructor. This is known as encapsulation. **Encapsulation is used to contain all of the functionalities of an object to isolate them within that object.**

```
Syntax. function Person (firstName, lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
            this.fullName = function() {return this.firstName + this.lastName;}
            };
            Person.prototype.greet = function () {        //To learn more about prototype see below
                  return "Hi, my name is " + fullName();
            }
Ex. function Car (make, mpg) {
      this.make = make;
      this.mpg = mpg;
            Car.prototype.pitch = function() {      //Now every instance of car will have this prototype
                  return "This care is a " + this.make + " and has " + this.mpg;
            }
      }
```

Accessing/Adding Properties in Objects

Note, if there is no such property in an object you will get undefined. For example, if you try to get the make property of the car object above you will get the value, however, if you try to get the model property you will get undefined.

Dot Notation

Dot notation only lets you access the explicit key name of a property. Can be used to add properties to objects.

```
Syntax: objectName.propertyName;          //Most common and easiest to use

Ex. var myObject = {
            key: value
};

myObject.key;          //Retains assigned value from the objects key-value pair. Accesses
myObject.property = value;       //Adds
```

Note: can use multiple "." To create sub-properties and values.
Note: Dot notation creates one at a time, left-to-right.

Bracket Notation

Javascript converts the expression between the brackets into a string and then retrieves the property that matches the string. Can be used to add properties to objects.

```
Syntax: objectName.propertyName = objectName["property"];

Ex. var myObject = {
            key: value
```

```
};

myObject["key"];            //Able to use part of property. Accesses
myObject["property"] = value;          //Adds
```

Object Inheritance, Prototypes and this

Objects have their own and inherited properties. Own properties are defined on the object i.e.: the properties you define for that object, whereas inherited properties are inherited from the objects prototype object. For example, any object I create using the Car constructor above will inherit the "pitch" prototype.

Methods

You can use methods to update an existing objects properties or calculate something using an objects properties. Methods are used by objects as functions. Some important methods to keep in mind are the hasOwnProperty and delete methods:

- You can use the hasOwnProperty method to check if an object has a property
  - syntax: object.hasOwnProperty(property); // returns true or false
- You can use the delete method to delete properties form an object
  - syntax: delete object.proprety; //returns true if the property was deleted
  - Cannot delete inherited properties this way. You must delete the prototype from the constructor

Creating Methods

Syntax: ObjectName.methodName = function(){
};

```
Ex. bob.setAge = function (newAge){      //Assume bob is an existing object
        bob.age = newAge;                //Creates setAge method
};
```

Calling a Method

We call a method like a function, with one difference. See below.

```
Ex. bob.setAge = function (newAge){      //Assume bob is an existing object
        bob.age = newAge;
};

objectName.methodName( );          //To call
```

Note: JS has a ton of built in methods, check out this site for a list: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Methods_Index

Prototype

Basically, a prototype is used for inheritance purposes. A prototype links back to the original object (parent) as a reference.

You can get the prototype of any object using the **getPrototype** method:

Syntax: getPrototypeOf(object);

Two things to keep in mind about prototypes:

- **Prototype Property**: Every function has a prototype property, which is empty by default, and you use it for inheritance purposes.
- **Prototype Attribute**: The attribute is used to point to the parent. Every object inherits properties from another object.
    - If you use object literal or constructor to create objects then they will inherit their properties fomr the Object.prototype.
    - If you create objects with the hybrid/function constructor then the object will inherit the properties you define

```
Syntax: className.prototype.newMethodName = function() {
            //do something;
};

Ex. function Dog (breed) {
        this.breed = breed;
};

Dog.prototype.bark = function() {
  console.log("Woof");
```

What is 'this'

Think of "this" as a pronoun. "this" is used to refer to the object that is being created. Every object (including functions) are created with a default "this" property. The "this" property always refers to and holds the value of the object that invokes the function where this is used.

Two things to keep in mind:

- Because "this" holds the value of the object when the function is invoked **it has no value until that happens.**
- Be careful when using "this" with closures and callbacks. It is important to note **that a closure will not be able to access the outer functions "this" value.**
    - To fix this problem set the "this" value to a SET value

      I.e: var storage = this;
        - now you can use the "this" value of the outer function by using the storage variable in the closure.

More to Follow

There is a whole other section to this guide taking place but I am not going to add it until I am sure that each portion above is fully developed. Let me know if you find any errors as this is a work in progress.

Thanks for reading,

Ryan Morris