**Variables**

* Variable names can contain letters, digits, underscores, and dollar signs.
* Variable names are case sensitive (y and Y are different variables)

**var**

A variable declared without a value will have the value undefined

You can just check if the variable has a truthy value or not. That means will evaluate to true if value is not:

* null
* undefined
* NaN
* empty string ("")
* 0
* false

**const**

Cannot be reassigned and not accessible before they appear within the code.

**let**

The let statement declares a block-scoped local variable, optionally initializing it to a value.

# Data Types

There are 8 basic data types in JavaScript.

There are essentially two types of values in JavaScript. The first type is primitives, and the second type is objects (which also includes functions). Primitive values include simple value types such as numbers (which includes everything from integers to floats to Infinity to NaN), booleans, strings, undefined, and null (note: even though typeof null === 'object', null is a still primitive value).

### number

numbers of any kind: integer or floating-point

**special numeric values**

* Infinity
* -Infinity
* NaN (Not a Number) - result of an incorrect or an undefined mathematical operation

### bigint

value is created by appending n to the end of an integer

### string

In JavaScript, there are 3 types of quotes.

1. Double quotes: "Hello".
2. Single quotes: 'Hello'.
3. Backticks: ` Hello `.

Double and single quotes are “simple” quotes. There’s practically no difference between them in JavaScript.

Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in ${…}

### boolean

### null

It’s just a special value which represents “nothing”, “empty” or “value unknown”.

### undefined

If a variable is declared, but not assigned

### object

Object and Array

### symbol

Every symbol value returned from Symbol() is unique.

A symbol value may be used as an identifier for object properties this is the data type's only purpose.

Is primary type so you can uses the new operator.  
var a = new Symbol(); // This will throw error.

# The typeof operator.

**typeof DATA**

### Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The typeof operator can return one of these primitive types:

* string
* number
* boolean
* undefined

### Complex Data

The typeof operator can return one of two complex types:

* function
* object

The typeof operator returns "object" for objects, arrays, and null.

The typeof operator does not return "object" for functions.

**String Methods**

* charAt - The String object's charAt() method returns a new string consisting of the single UTF-16 code unit located at the specified offset into the string. Return Character.  
  **STRING1.charAt(INDEX)**
* charCodeAt - The charCodeAt() method returns an integer between 0 and 65535 representing the UTF-16 code unit at the given index. Return Number.  
  **STRING1.charCodeAt(INDEX)**
* concat - joins two or more strings. Return String.  
  **STRING1.concat(STRING2)**
* constructor - The String constructor is used to create a new String object. When called instead as a function, it performs type conversion to a primitive string, which is usually more useful. Return function.  
  **STRING1.constructor**
* endsWith - Check if a string ends with. Return boolean.  
  **STRING1.endsWith(VALUE)**
* fromCharCode - Convert a Unicode number into a character. Return character.  
  **String.fromCharCode(NUMBER)**.
* includes - Check if a string includes certain string. Return boolean.  
  **STRING1.includes(VALUE)**
* indexOf - Search a certain String and return index of position of the first character in the Character array. Return number.  
  **STRING1.indexOf(VALUE)**
* lastIndexOf - Search a string for the last occurrence of certain String. Return index of position of the first character in the Character array. Return number.  
  **STRING1.lastIndexOf(VALUE)**
* length - Return the number of characters in a string. Return number. **STRING1.length**
* localeCompare - A Number, indicating whether the reference string comes before, after or is the same as the compareString in sort order. Returns one of three number values:  
  **STRING1.localeCompare(STRING2)**
* a) -1 if the reference string is sorted before the compareString
* b) 0 if the two strings are equal
* c) 1 if the reference string is sorted after the compareString
* match - Search for the same pattern in string using Regular Expressions. Return array.  
  **STRING1.match(REGULAR\_EXPRESSIONS)**
* prototype - Use the prototype property to add a new property to all objects of a given type.  
  **FUNCTION1.prototype.VALUE**
* repeat - Make a new string by copying the original string number of times. Return string.  
  **STRING1.repeat(VALUE)**.
* replace - The replace method returns a new string with some or all matches of a pattern replaced by a replacement. The pattern can be a string or a RegExp, and the replacement can be a string or a function to be called for each match. If pattern is a string, only the first occurrence will be replaced. Return string.  
  **STRING1.replace(REGULAR\_EXPRESSIONS, REPLACE\_VALUE)**
* search - The search method executes a search for a match between a regular expression and this String object. Return index position of the search string. Return number.  
  **STRING1.search(VALUE)**
* slice - The slice method returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) where start and end represent the index of items in that array. The original array will not be modified. Return string.  
  **STRING1.slice(start, end)**
* split - The split method divides a String into an ordered list of substrings, puts these substrings into an array, and returns the array. The division is done by searching for a pattern; where the pattern is provided as the first parameter in the method's call. Return array.  
  **STRING1.split(separator, limit)**
* startsWith - The startsWith method determines whether a string begins with the characters of a specified string, returning true or false as appropriate. Return boolean.  
  **STRING1.startsWith(searchvalue, start)**
* substr - The substr method returns a portion of the string, starting at the specified index and extending for a given number of characters afterwards. Return string.  
  **STRING1.substr(start, length)**
* substring - The substr method returns a portion of the string, starting at the specified index and extending for a given number of characters afterwards. Return string.  
  **STRING1.substring(start, end)**
* toLocaleLowerCase - Convert the string to lowercase letters. Return string.  
  **STRING1.toLocaleLowerCase()**
* toLocaleUpperCase - Convert the string to uppercase letters. Return string.  
  **STRING2.toLocaleUpperCase()**
* toLowerCase - Convert the string to lowercase letters. Return string.  
  **STRING1.toLowerCase()**
* toString - Return the value of a string. Return string.  
  **STRING1.toString()**
* toUpperCase - Convert the string to uppercase letters. Return string.  
  **STRING1.toUpperCase()**
* trim - Remove whitespace from both sides of a string. Return string.  
  **STRING1.trim()**
* valueOf - Return the primitive value of a string object. Return string.  
  **STRING1.valueOf()**

# Date

There are four ways of instantiating a date:  
var d = new Date();  
var d = new Date(milliseconds);  
var d = new Date(dateString);  
var d = new Date(year, month, day, hours, minutes, seconds, milliseconds);

# Arrays Metods

* concat - Join several arrays into one. Return array.  
  **ARRAY1.concat(ARRAY2)**
* constructor The Array() constructor is used to create Array objects.  
  **ARRAY1.constructor**
* copyWithin - Copies array elements to another position in an array, overwriting the existing values. Return array.  
  **ARRAY1.copyWithin(target, start, end)**
* entries- Array Iterator object that contains the key/value pairs for each index in the array. Return key/value pairs array.  
  **for( var [index, element] of ARRAY1.entries())( console.log(index, element);}**
* every - The every method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value. Return boolean.  
  **ARRAY1.every(function(value){ return value; } )**
* fill - Fill all the array elements with a static value. Return array.  
  **ARRAY1.fill(VALUE)**
* filter - The filter method creates a new array with all elements that pass the test implemented by the provided function. Return array.  
  **ARRAY1.filter(function(value){ return value; } )**
* find - Give a value this method will find the value. Return number.  
  **ARRAY1.find(function(value){ return value; } )**
* findIndex- Give a value this method will find index for that value. Return number.  
  **ARRAY1.findIndex(function(value){ return value; } )**
* forEach - List each item in the array. Return array.  
  **ARRAY1.forEach(function(item, index){ console.log( index + " " + item) } )**
* from - Create an Array from another array. Pass in function to change initial array. Return array.  
  **var mult = { do(x) { return x this.factor; } };**  
  **Array.from(ARRAY1, mult.do, {factor:2} )**
* includes - Check if an array includes certain value. Return boolean.  
  **ARRAY1.includes(VALUE)**
* indexOf - Search an array for the item and return the index of that value. Return number.  
  **Array1.indexOf(VALUE)**
* isArray - Check whether an object is an array. Return boolean.  
  **Array.isArray(ARRAY1)**
* join - Convert the elements of an array into a string. Return string.  
  **ARRAY1.join()**
* keys - Create an Array Iterator object, containing the keys of the array. Return array.  
  **ARRAY1.keys()**
* length - Return the length of an array. Return number.  
  **ARRAY1.length**
* lastIndexOf - Search an array for the item. If there are many times certain item is in the array will return last index. Return number.  
  **ARRAY1.lastIndexOf(VALUE)**
* map - A function to be run for each element in the array. Return array.  
  **function multBy(currentValue, index, arr){ return currentValue\*this.factor; }**  
  **ARRAY1.map(multBy, {factor,2})**
* pop - Remove the last element of an array. Return element.  
  **ARRAY1.pop()**
* prototype - Create a method on the array.
* push - Add a new item to an array.  
  **ARRAY1.push(VALUE)**
* reduce - The reduce method executes a user-supplied “reducer” callback function on each element of the array, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value. Return number of total value.  
  **function mult((total, currentValue, index, arr){ return total + currentValue; //Calculate total }**.  
  **ARRAY1.reduce(mult, INITIAL\_VALUE\_FOR\_TOTAL)**
* reduceRight - The reduceRight method applies a function against an accumulator and each value of the array **(from right-to-left)** to reduce it to a single value.  
  **function mult((total, currentValue, index, arr){ return total + currentValue; //Calculate total }**.  
  **ARRAY1.reduce(mult, INITIAL\_VALUE\_FOR\_TOTAL)**
* reverse - Reverse the order of the elements in an array. Return array.  
  **ARRAY1.reverse()**
* shift - Remove the first item of an array. Return element from array.  
  **ARRAY1.shift()**
* slice - The slice method returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) where start and end represent the index of items in that array. The original array will not be modified. Return array.  
  **ARRAY1.slice(startIndex, endIndex)**
* some - The some method tests whether at least one element in the array passes the test implemented by the provided function. It returns true if, in the array, it finds an element for which the provided function returns true; otherwise it returns false. It doesn't modify the array. Return boolean.  
  **function checkValue(num){ return num > 10; }**  
  **ARRAY1.some(checkvalue))**
* sort - The sort method sorts the elements of an array. Return array.  
  **ARRAY1.sort(function(a, b){return a-b}) )**
* splice - The splice method changes the contents of an array by removing or replacing existing elements and/or adding new elements. Return array.  
  **ARRAY1.splice(startIndex, numberElementToReplace, newElements1, newElements2,...)**.
* toString - The unshift method adds one or more elements to the beginning of an array and returns the new length of the array. Return string.  
  **ARRAY1.toString()**
* unshift - he unshift method adds one or more elements to the beginning of an array and returns the new length of the array. Return array.  
  **ARRAY1.unshift(newElements1, newElements2,...)**
* valueOf - The valueOf method returns the primitive value of the specified object.

**Functions**

**Types of functions**

* Constructor - **var add = new Function( 'num1', 'num2', 'return num1 + num2')**
* Declaration - **function mult(x,y){ return x\*y; }**
* Expression - **var substract = function(x,y){ return x-y;}**

**IIFE (Immediately Invoked Function Expression)**

An Immediately-invoked Function Expression is a way to execute functions immediately, as soon as they are created.

**result = (function(a, b){return a - b;})(100, 42)**

**Closure**

In JavaScript, a closure is a function that references variables in the outer scope from its inner scope. The closure preserves the outer scope inside its inner scope.

**function makeCounter(){ var i=0; return function(){ +i }; }**

**Function in Object**

**banana ={}**.  
**banana.count = function(){ return 0; }**  
**var fruits = { 'banana': banana.count }**

**Two ways to call a function**

* **fruits[fruit\_name]()**
* **fruits.bananas()**

**Callback Function**

You know the parameters are for callback function but have not wrote the callback function until later.

# Regular Expressions

There are two ways to create a RegExp object: a literal notation and a constructor.

* The first and last / is where the Regular Expression starts and ends.

### Literal Notation

/pattern/modifiers;

| **Parameter** | **Description** |
| --- | --- |
| pattern | Required. A regular expression is an object that describes a pattern of characters. |
| modifier | Optional. Modifiers are used to change perform of the Regular Expression. |

### Constructor

new RegExp(pattern, modifier)

| **Parameter** | **Description** |
| --- | --- |
| pattern | Required. A regular expression is an object that describes a pattern of characters. |
| modifier | Optional. Modifiers are used to change perform of the Regular Expression. |

### Regular Expressions Methods

* RegExp.prototype.**exec** - The exec method executes a search for a match in a specified string. Returns a result array, or null.
* RegExp.prototype.**test** - The test method executes a search for a match between a regular expression and a specified string.
* String.prototype.**match** - The match method retrieves the result of matching a string against a regular expression.
* String.prototype.**matchAll** - The matchAll method returns an iterator of all results matching a string against a regular expression, including capturing groups.
* String.prototype.**replace** - The replace method returns a new string with some or all matches of a pattern replaced by a replacement.
* String.prototype.**replaceAll** - The replaceAll method returns a new string with all matches of a pattern replaced by a replacement.
* String.prototype.**search** - The search method executes a search for a match between a regular expression and this String object.

## Modifiers

* **d Flag** - The "d" flag indicates that the result of a regular expression match should contain the **start and end indices** of the substrings of each capture group. **Corresponding property** RegExp.prototype.hasIndices
* **g Flag** - The "g" flag indicates that the regular expression should be tested **against all possible matches** in a string. A regular expression defined as both global ("g") and sticky ("y") will ignore the global flag and perform sticky matches. **Corresponding property** RegExp.prototype.hasIndices
* **i Flag** - Do a **case-insensitive** search. **Corresponding property** RegExp.prototype.ignoreCase
* **m Flag** - Multi-line search. **Corresponding property** RegExp.prototype.multiline
* **s Flag** - Allows to match newline characters. **Corresponding property** RegExp.prototype.dotAll
* **u Flag** - "unicode"; treat a pattern as a sequence of unicode code points. **Corresponding property** RegExp.prototype.unicode
* **y Flag** - Perform a "sticky" search that matches starting at the current position in the target string. **Corresponding property** RegExp.prototype.sticky

## Groups and ranges

* **(x|y)** - Matches either "x" or "y". For example, /green|red/ matches "green" in "green apple" and "red" in "red apple".
* **[xyz] or [a-c]** - A character class. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character class as a normal character. For example, [abcd] is the same as [a-d]. They match the "b" in "brisket", and the "c" in "chop". For example, [abcd-] and [-abcd] match the "b" in "brisket", the "c" in "chop", and the "-" (hyphen) in "non-profit".
* **[^xyz] or [^a-c]** - A negated or complemented character class. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character class as a normal character.  
  For example, [^abc] is the same as [^a-c]. They initially match "o" in "bacon" and "h" in "chop".
* **(x)** - Capturing group: Matches x and remembers the match. For example, /(foo)/ matches and remembers "foo" in "foo bar".
* **\n** -Where "n" is a positive integer. \1 refers to the first capturing group in the regular expression. \2 will refer to the second capturing group and \n will refer to an nth capturing group.  
  Where "n" is a positive integer. A back reference to the last substring matching the n parenthetical in the regular expression (counting left parentheses). For example, /apple(,)\sorange\1/ matches "apple, orange," in "apple, orange, cherry, peach".

## Meta characters

* **.** - Find a single character, except newline or line terminator.
* **\w** - Find a word character. A word character is a character a-z, A-Z, 0-9, including \_ (underscore).
* **\W** - Find a non-word character.
* **\d** - Find a digit.
* **\D** - Find a non-digit character.
* **\s** - Find a whitespace character.
* **\S** - Find a non-whitespace character.
* **\b** - Find a match at the beginning/end of a word, beginning like this: \bHI, end like this: HI\b.
* **\B** - Find a match, but not at the beginning/end of a word.
* **\0** - Find a NULL character.
* **\n** - Find a new line character.
* **\f** - Find a form feed character.
* **\r** - Find a carriage return character.
* **\t** - Find a tab character.
* **\v** - Find a vertical tab character.
* **\xxx** - Find the character specified by an octal number xxx.
* **\xdd** - Find the character specified by a hexadecimal number dd.
* **\udddd** - Find the Unicode character specified by a hexadecimal number dddd.

## Quantifiers

* **n+** - Matches any string that contains **at least one** n.
* **n**\* - Matches any string that contains **zero or more** occurrences of n.
* **n?** - Matches any string that contains **zero or one** occurrences of n.
* **n{X}** - Matches any string that contains **a sequence of X n's**.
* **n{X,Y}** - Matches any string that contains **a sequence of X to Y n's**.
* **n{X,}** - Matches any string that contains **a sequence of at least X n's**.
* **n$** - Matches any string with n at the **end** of it.
* **^n** - Matches any string with n at the **beginning** of it.
* **?=n** - Matches any string that is **followed by a specific** string n.
* **?!n** - Matches any string that is not **followed by a specific** string n.

# Callback

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.