Javascript Cheatsheet

# Regular Expressions (Regex)

## Using the Test Method: /regex/.test('string') looks for a specific word in a string.

**Match Literal Strings:** 'string'.match(/regex/)

**Match a Literal String with Different Possibilities:** This is powerful to search single strings, but it's limited to only one pattern. You can search for multiple patterns using the alternation or OR operator: |. e.g.: /yes|no|maybe/.

**Ignore Case While Matching:** /ignorecase/i // use lowercase “i” outside of /’s.

One or more: Use “+” to look for one or more characters.

0 or more: Use “ \* ” (asterisk) to look for zero or more characters.

**Extract Matches:** use the .match() method at the end of a string variable to get exact matches. The exact word or phrase is placed between the ().Note that the .match syntax is the "opposite" of the .test method you have been using thus far: 'string'.match(/regex/); /regex/.test('string');

**Find More Than the First Match:** To search or extract a pattern more than once, you can use the g flag. e.g. : /Repeat/g. Similar to the “i” flag.

**Match Beginning String Patterns**: wildcard character: “.” The dot, used like: /hu./;

**Match Single Character with Multiple Possibilities**: You can search for a literal pattern with some flexibility with character classes. Character classes allow you to define a group of characters you wish to match by placing them inside square ([ and ]) brackets. e.g. /b[aiu]g/

**Match Letters of the Alphabet**: match lowercase letters a through e you would use [a-e]. Matches every word in the bracket.

**Match Numbers and Letters of the Alphabet:** Using the hyphen (-) to match a range of characters is not limited to letters. It also works to match a range of numbers. e.g.: /[a-z0-9]/**ig**

**Match Single Characters Not Specified**: To create a negated character set, you place a caret character (^) after the opening bracket and before the characters you do not want to match.

For example, /[^aeiou]/gi matches all characters that are not a vowel. Note that characters like ., !, [, @, / and white space are matched - the negated vowel character set only excludes the vowel characters.

**Match Characters that Occur One or More Times**: match a character (or group of characters) that appears one or more times in a row. This means it occurs at least once, and may be repeated. e.g. /a+/g

**Match Characters that Occur Zero or More Times:** the plus + sign to look for characters that occur one or more times. There's also an option that matches characters that occur zero or more times. The character to do this is the asterisk or star: “\*”. /go\*/

**Find Characters with Lazy Matching:** use the ? character to change it to lazy matching. "titanic" matched against the adjusted regex of /t[a-z]\*?i/ returns ["ti"].

**Find One or More Criminals in a Hunt:** The regex /z+/ matches the letter z when it appears one or more times in a row.

**Match Beginning String Patterns**: In an earlier challenge, you used the caret character (^) inside a character set to create a negated character set in the form [^thingsThatWillNotBeMatched]. Outside of a character set, the caret is used to search for patterns at the beginning of strings.

**Match Ending String Patterns**: search the end of strings using the dollar sign character $ at the end of the regex. /story$/;

**Match All Letters and Numbers:** Using character classes, you were able to search for all letters of the alphabet with [a-z]. This kind of character class is common enough that there is a shortcut for it, although it includes a few extra characters as well. The closest character class in JavaScript to match the alphabet is \w. This shortcut is equal to [A-Za-z0-9\_]. This character class matches upper and lowercase letters plus numbers. Note, this character class also includes the underscore character (\_). e.g.: /\w/

**Match Everything But Letters and Numbers**: You can search for the opposite of the \w with \W. Note, the opposite pattern uses a capital letter. This shortcut is the same as [^A-Za-z0-9\_].

**Match All Numbers:** Use the shorthand character class \d to count how many digits are in movie titles. Written out numbers ("six" instead of 6) do not count. /\d/g

**Match All Non-Numbers:** Use the shortcut to look for non-digit characters is \D. This is equal to the character class [^0-9], which looks for a single character that is not a number between zero and nine. /\D/g

**Restrict Possible Usernames**: Usernames are used everywhere on the internet. They are what give users a unique identity on their favorite sites.

You need to check all the usernames in a database. Here are some simple rules that users have to follow when creating their username.

1) Usernames can only use alpha-numeric characters.

2) The only numbers in the username have to be at the end. There can be zero or more of them at the end. Username cannot start with the number.

3) Username letters can be lowercase and uppercase.

4) Usernames have to be at least two characters long. A two-character username can only use alphabet letters as characters.

Solution is : /^[a-z]([0-9][0-9]+|[a-z]+\d\*)$/i or /^[a-z]([0-9]{2,}|[a-z]+\d\*)$/i

**Match Whitespace:** The challenges so far have covered matching letters of the alphabet and numbers. You can also match the whitespace or spaces between letters. You can search for whitespace using \s, which is a lowercase s. This pattern not only matches whitespace, but also carriage return, tab, form feed, and new line characters. You can think of it as similar to the character class [ \r\t\f\n\v]. e.g.: let spaceRegex = /\s/g;

**Match Non-Whitespace Characters:** You learned about searching for whitespace using \s, with a lowercase s. You can also search for everything except whitespace. Search for non-whitespace using \S, which is an uppercase s. This pattern will not match whitespace, carriage return, tab, form feed, and new line characters. You can think of it being similar to the character class [^ \r\t\f\n\v].

e.g.:

let whiteSpace = "Whitespace. Whitespace everywhere!"

let nonSpaceRegex = /\S/g;

whiteSpace.match(nonSpaceRegex).length; // Returns 32

**Specify Upper and Lower Number of Matches:** Recall that you use the plus sign + to look for one or more characters and the asterisk \* to look for zero or more characters. These are convenient but sometimes you want to match a certain range of patterns. You can specify the lower and upper number of patterns with quantity specifiers. Quantity specifiers are used with curly brackets ({ and }). You put two numbers between the curly brackets - for the lower and upper number of patterns.

For example, to match the entire phrase "Oh no" only when it has 3 to 6 letter h's..

let ohStr = "Ohhh no";

let ohRegex = /Oh{3,6}\sno/; // Change this line \s to include white spaces.

let result = ohRegex.test(ohStr);

**Specify Only the Lower Number of Matches:** You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want to specify the lower number of patterns with no upper limit.

To only specify the lower number of patterns, keep the first number followed by a comma.

For example, to match only the string "hah" with the letter a appearing at least 3 times, your regex would be /ha{3,}h/. Take string and replace repeating letter with repeat\_letter{x,} for x times.

**Specify Exact Number of Matches:** You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want a specific number of matches. To specify a certain number of patterns, just have that one number between the curly brackets.

For example, to match only the word "hah" with the letter a 3 times, your regex would be /ha{3}h/.

**Check for All or None:** Sometimes the patterns you want to search for may have parts of it that may or may not exist. However, it may be important to check for them nonetheless. You can specify the possible existence of an element with a question mark, ?. This checks for zero or one of the preceding element. You can think of this symbol as saying the previous element is optional.

For example, there are slight differences in American and British English and you can use the question mark to match both spellings.

let american = "color";

let british = "colour";

let rainbowRegex= /colou?r/; // use the british version to search.

rainbowRegex.test(american); // Returns true

rainbowRegex.test(british); // Returns true

**Positive and Negative Lookahead:** Lookaheads are patterns that tell JavaScript to look-ahead in your string to check for patterns further along. This can be useful when you want to search for multiple patterns over the same string.

There are two kinds of lookaheads: positive lookahead and negative lookahead.

A positive lookahead will look to make sure the element in the search pattern is there, but won't actually match it. **A positive lookahead is used as (?=...)** where the ... is the required part that is not matched.

On the other hand, a negative lookahead will look to make sure the element in the search pattern is not there. **A negative lookahead is used as (?!...)** where the ... is the pattern that you do not want to be there. The rest of the pattern is returned if the negative lookahead part is not present.

Lookaheads are a bit confusing but some examples will help.

let quit = "qu";

let noquit = "qt";

let quRegex= /q(?=u)/;

let qRegex = /q(?!u)/;

quit.match(quRegex); // Returns ["q"]

noquit.match(qRegex); // Returns ["q"]

A more practical use of lookaheads is to check two or more patterns in one string. Here is a (naively) simple password checker that looks for between 3 and 6 characters and at least one number:

let password = "abc123";

let checkPass = /(?=\w{3,6})(?=\D\*\d)/;

checkPass.test(password); // Returns true

Reuse Patterns Using Capture Groups:

Some patterns you search for will occur multiple times in a string. It is wasteful to manually repeat that regex. There is a better way to specify when you have multiple repeat substrings in your string.

You can search for repeat substrings using *capture groups*. Parentheses, ( and ), are used to find repeat substrings. You put the regex of the pattern that will repeat in between the parentheses.

To specify where that repeat string will appear, you use a backslash (\) and then a number. This number starts at 1 and increases with each additional capture group you use. An example would be \1 to match the first group.

The example below matches any word that occurs twice separated by a space:

let repeatStr = "regex regex";

let repeatRegex = /(\w+)\s\1/;

repeatRegex.test(repeatStr);

repeatStr.match(repeatRegex);

For challenge: / ^(\d+)\s\1\s\1$/

Check For Mixed Grouping of Characters:

**Flag g: looks for repeat strings, /Repeat/g**

**Flag i: ignores case,** /ignorecase/i

**Match a Literal String with Different Possibilities: “|”** e.g.: /yes|no|maybe/

**Match Beginning String Patterns**: “.”

**Match Single Characters Not Specified: ^**

**Match Beginning String Patterns**: ^

**Match Ending String Patterns**: $

**Match ALL Letters and Numbers**: \w

**Match Everything But Letters and Numbers**: \W

**Match All Numbers**: \d

**Match All Non-Numbers: \D**

**Match Whitespace:** \s

**Match Non-Whitespace Characters:** \S

**Specify Upper and Lower Number of Matches:** {3,5}

**Specify Only the Lower Number of Matches: {3,}**

**Specify Exact Number of Matches:** {3}

**Check for All or None:** ?

**plus sign + to look for one or more characters**

**the asterisk \* to look for zero or more characters**

**A positive lookahead is used as (?=...)** where the ... is the required part that is not matched.

**A negative lookahead is used as (?!...)** where the ... is the pattern that you do not want to be there.

A “.” Is a search wildcard.

.replace(/(^|\s)\S/g,

L => L.toUpperCase().

**/(?:^|\s|["'([{])+\S/g**

# Data structures:

methods take one or more elements as parameters and add those elements to the array the method is being called on;

.split() method converts strings to an array.

.join() method converts array to strings.

push() method adds elements to the end of an array,

unshift() adds elements to the beginning.

pop() removes an element from the end of an array,

shift() removes an element from the beginning.

The key difference between pop() and shift() and their cousins push() and unshift(), is that neither method takes parameters, and each only allows an array to be modified by a single element at a time.

splice() can take up to 3 parameters. The first two parameters of splice() are integers which represent indexes, or positions, of the array that splice() is being called upon to be removed. The third parameter, comprised of one or more element(s), to add to the array.

numbers.splice(startIndex, amountToDelete, 1itemToAdd, 2itemToAdd, … nth itemToAdd);

slice(), rather than modifying an array, copies, or extracts, a given number of elements to a new array, leaving the array it is called upon untouched.

slice() takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (extraction will occur up to, but not including the element at this index).

Since arrays can be changed, or mutated, at any time, there's no guarantee about where a particular piece of data will be on a given array, or if that element even still exists. Luckily, JavaScript provides us with another built-in method, indexOf(), that allows us to quickly and easily check for the presence of an element on an array. *indexOf() takes an element as a parameter, and when called, it returns the position, or index, of that element, or -1 if the element does not exist on the array.*

For example:

let fruits = ['apples', 'pears', 'oranges', 'peaches', 'pears'];

fruits.indexOf('dates'); // returns -1

fruits.indexOf('oranges'); // returns 2

fruits.indexOf('pears'); // returns 1, the first index at which the element exists

indexOf() can be incredibly useful for quickly checking for the presence of an element on an array.

There are two ways to access an individual character in a string. The first is the .[charAt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/charAt) method:

The other way (introduced in ECMAScript 5) is to treat the string as an array-like object, where individual characters correspond to a numerical index:

return 'cat'[1] // returns "a"

# String manipulation methods:

The .test() method takes the regex, applies it to a string (which is placed inside the parentheses), and returns true or false if your pattern finds something or not.

The toLowerCase() method returns the calling string value converted to lowercase

The split() method splits a String object into an array of strings by separating the string into substrings.

The charAt() method returns the specified character from a string.

The slice() method extracts a section of a string and returns a new string.

The join() method joins all elements of an array into a string.

The map() method creates a new array with the results of calling a provided function on every element in this array. Using map will call a provided callback function once for each element in an array, in order, and constructs a new array from the results.

The replace() method returns a new string with some or all matches of a pattern replaced by a replacement.

# Resources

* [toLowerCase() method — MDN](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase)
* [toUpperCase() method — MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase)
* [charAt() method — MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/charAt)
* [slice() method — MDN](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/String/slice)
* [split() method — MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split)
* [join() method — MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join)
* [for — MDN](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/for)
* [map() method — MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)
* [replace() method — MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace)

arr.sort((a, b) => a - b); sorts an array from lowest to highest values. (numbers);

## Object Oriented Design:

**this** refers to the object that the method is associated with; the objects name.

Constructors follow a few conventions:

* Constructors are defined with a capitalized name to distinguish them from other functions that are not constructors.
* Constructors use the keyword this to set properties of the object they will create. Inside the constructor, this refers to the new object it will create.
* Constructors define properties and behaviors instead of returning a value as other functions might.

Anytime a constructor function creates a new object, that object is said to be an ***instance*** of its constructor. JavaScript gives a convenient way to verify this with the *instanceof* operator. *instanceof* allows you to compare an object to a constructor, returning true or false based on whether or not that object was created with the constructor.

Use Prototype properties to reduce duplicate code:

object.prototype.redundantproperty = value for property;

i.e. : Bird.prototype.numLegs = 2;

Since all instances automatically have the properties on the prototype, think of a prototype as a *"recipe"* for creating objects. Note that the prototype for duck and canary is part of the Bird constructor as *Bird.prototype*. Nearly every object in JavaScript has a prototype property which is part of the constructor function that created it.

You have now seen two kinds of properties: own properties and prototype properties. Own properties are defined directly on the object instance itself. And prototype properties are defined on the prototype.

let ownProps = [];

let prototypeProps = [];

for (let property in duck) {

if(duck.hasOwnProperty(property)) {

ownProps.push(property);

} else {

prototypeProps.push(property);

}

}

console.log(ownProps); // prints ["name"]

console.log(prototypeProps); // prints ["numLegs"]

There is a special constructor property located on the object instance. Note that the constructor property is a reference to the constructor function that created the instance. The advantage of the constructor property is that it's possible to check for this property to find out what kind of object it is.

Note  
Since the constructor property can be overwritten, it’s generally better to use the instanceof method to check the type of an object.

There is one crucial side effect of manually setting the prototype to a new object. It erases the constructor property! This property can be used to check which constructor function created the instance, but since the property has been overwritten, it now gives false results:

duck.constructor === Bird; // false -- Oops

duck.constructor === Object; // true, all objects inherit from Object.prototype

duck instanceof Bird; // true, still works

To fix this, whenever a prototype is manually set to a new object, remember to define the constructor property:

Bird.prototype = {

constructor: Bird, // define the constructor property

numLegs: 2,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

Inheritance:

 The first step: make an instance of the supertype (or parent).

let animal = Object.create(Animal.prototype);

e.g. : let duck = Object.create(Animal.prototype);

## Set the Child's Prototype to an Instance of the Parent

Set the Child's Prototype to an Instance of the Parent:

the next step: set the prototype of the subtype (or child)—in this case, Bird—to be an instance of Animal.

Bird.prototype = Object.create(Animal.prototype);

Reset an Inherited Constructor Property:

When an object inherits its prototype from another object, it also inherits the supertype's constructor property.

Here's an example:

function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

let duck = new Bird();

duck.constructor // function Animal(){...}

But duck and all instances of Bird should show that they were constructed by Bird and not Animal. To do so, you can manually set Bird's constructor property to the Bird object:

Bird.prototype.constructor = Bird;

duck.constructor // function Bird(){...}

An object can inherit its behavior (methods) from another object by referencing its prototype object:

ChildObject.prototype = Object.create(ParentObject.prototype);

Then the ChildObject received its own methods by chaining them onto its prototype:

ChildObject.prototype.methodName = function() {...};

It's possible to override an inherited method. It's done the same way - by adding a method to ChildObject.prototype using the same method name as the one to override.

e.g.:

function Animal() { }

Animal.prototype.eat = function() {

return "nom nom nom";

};

function Bird() { }

// Inherit all methods from Animal

Bird.prototype = Object.create(Animal.prototype);

// Bird.eat() overrides Animal.eat()

Bird.prototype.eat = function() {

return "peck peck peck";

};

If you have an instance let duck = new Bird(); and you call duck.eat(), this is how JavaScript looks for the method on duck’s prototype chain:

1. duck => Is eat() defined here? No.
2. Bird => Is eat() defined here? => Yes. Execute it and stop searching.
3. Animal => eat() is also defined, but JavaScript stopped searching before reaching this level.
4. Object => JavaScript stopped searching before reaching this level.

Use a Mixin to Add Common Behavior Between Unrelated Objects:

As you have seen, behavior is shared through inheritance. However, there are cases when inheritance is not the best solution. Inheritance does not work well for unrelated objects like Bird and Airplane. They can both fly, but a Bird is not a type of Airplane and vice versa.

For unrelated objects, it's better to use mixins. A mixin allows other objects to use a collection of functions.

let flyMixin = function(obj) {

obj.fly = function() {

console.log("Flying, wooosh!");

}

};

The flyMixin takes any object and gives it the fly method.

let bird = {

name: "Donald",

numLegs: 2

};

let plane = {

model: "777",

numPassengers: 524

};

flyMixin(bird);

flyMixin(plane);

Use Closure to Protect Properties Within an Object from Being Modified Externally:

In JavaScript, a function always has access to the context in which it was created. This is called closure.

To set variable private, set “this.variable” to let variable. This sets the variable local to the object method.

Then create a method to retrieve variable: this.getObject = function() { return variable};

Understand the Immediately Invoked Function Expression (IIFE)

A common pattern in JavaScript is to execute a function as soon as it is declared:

(function () {

console.log("Chirp, chirp!");

}) (); // this is an anonymous function expression that executes right away

// Outputs "Chirp, chirp!" immediately

Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an *immediately invoked function expression* or IIFE.

(function () {

  console.log("A cozy nest is ready");

})(); // it's an anonymous immediately invoked function expression (IIFE).

Use an IIFE to Create a Module:

An immediately invoked function expression (IIFE) is often used to group related functionality into a single object or module.

# Functional Programming:

## Understand Functional Programming Terminology

The FCC Team had a mood swing and now wants two types of tea: green tea and black tea. General Fact: Client mood swings are pretty common.

With that information, we'll need to revisit the getTea function from last challenge to handle various tea requests. We can modify getTea to accept a function as a parameter to be able to change the type of tea it prepares. This makes getTea more flexible, and gives the programmer more control when client requests change.

But first, let's cover some functional terminology:

Callbacks are the functions that are slipped or passed into another function to decide the invocation of that function. You may have seen them passed to other methods, for example in filter, the callback function tells JavaScript the criteria for how to filter an array.

Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value, are called first class functions. In JavaScript, all functions are first class functions.

The functions that take a function as an argument, or return a function as a return value are called higher order functions.

When the functions are passed in to another function or returned from another function, then those functions which gets passed in or returned can be called a lambda.

## Understand the Hazards of Using Imperative Code:

Functional programming is a good habit. It keeps your code easy to manage, and saves you from sneaky bugs. But before we get there, let's look at an imperative approach to programming to highlight where you may have issues.

In English (and many other languages), the imperative tense is used to give commands. Similarly, an imperative style in programming is one that gives the computer a set of statements to perform a task.

## Use the map Method to Extract Data from an Array

So far we have learned to use pure functions to avoid side effects in a program. Also, we have seen the value in having a function only depend on its input arguments.

This is only the beginning. As its name suggests, functional programming is centered around a theory of functions.

It would make sense to be able to pass them as arguments to other functions, and return a function from another function. Functions are considered first class objects in JavaScript, which means they can be used like any other object. They can be saved in variables, stored in an object, or passed as function arguments.

Let's start with some simple array functions, which are methods on the array object prototype. In this exercise we are looking at Array.prototype.map(), or more simply map.

The map method iterates over each item in an array and returns a new array containing the results of calling the callback function on each element. It does this without mutating the original array.

When the callback is used, it is passed three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the map method was called.

See below for an example using the map method on the users array to return a new array containing only the names of the users as elements. For simplicity, the example only uses the first argument of the callback.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const names = users.map(user => user.name);

console.log(names); // [ 'John', 'Amy', 'camperCat' ]

Often the statements change the state of the program, like updating global variables. A classic example is writing a for loop that gives exact directions to iterate over the indices of an array.

In contrast, functional programming is a form of declarative programming. You tell the computer what you want done by calling a method or function.

JavaScript offers many predefined methods that handle common tasks so you don't need to write out how the computer should perform them. For example, instead of using the for loop mentioned above, you could call the map method which handles the details of iterating over an array. This helps to avoid semantic errors, like the "Off By One Errors" that were covered in the Debugging section.

Consider the scenario: you are browsing the web in your browser, and want to track the tabs you have opened. Let's try to model this using some simple object-oriented code.

A Window object is made up of tabs, and you usually have more than one Window open. The titles of each open site in each Window object is held in an array. After working in the browser (opening new tabs, merging windows, and closing tabs), you want to print the tabs that are still open. Closed tabs are removed from the array and new tabs (for simplicity) get added to the end of it.

The code editor shows an implementation of this functionality with functions for tabOpen(), tabClose(), and join(). The array tabs is part of the Window object that stores the name of the open pages.

An **arrow function expression** is a syntactically compact alternative to a regular [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function), although without its own bindings to the [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), or [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target) keywords. Arrow function expressions are ill suited as methods, and they cannot be used as constructors.

var variableName = objectList

  .map( varName => {

    return{

      resultName1: varName.quotedItemInObj1,

      resultName2: varName.quotedItemInObj2

    }

  })

Will return:

[{"resultName1":"valueOfObj1"," quotedItemInObj1 ":" valueOfObj "},

{" resultName2":" valueOfObj2"," quotedItemInObj2":" valueOfObj "}]

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write.

JavaScript Object Notation is an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types.

<https://en.wikipedia.org/wiki/JSON>

**Use the reduce Method to Analyze Data**

**Array.prototype.reduce(), or simply reduce(), is the most general of all array operations in JavaScript. You can solve almost any array processing problem using the reduce method.**

**The reduce method allows for more general forms of array processing, and it's possible to show that both filter and map can be derived as special applications of reduce. The reduce method iterates over each item in an array and returns a single value (i.e. string, number, object, array). This is achieved via a callback function that is called on each iteration.**

**The callback function accepts four arguments. The first argument is known as the accumulator, which *gets assigned the return value of the callback function from the previous iteration*, the second is the current element being processed, the third is the index of that element and the fourth is the array upon which reduce is called.**

**In addition to the callback function, reduce has an additional parameter which takes an initial value for the accumulator. If this second parameter is not used, then the first iteration is skipped and the second iteration gets passed the first element of the array as the accumulator.**

**See below for an example using reduce on the users array to return the sum of all the users' ages. *For simplicity, the example only uses the first and second arguments*.**

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

(sum, user) => sum + user.age = is the “accumulator”

0 is the current element being processed

const sumOfAges = users.reduce((sum, user) => sum + user.age, 0);

console.log(sumOfAges); // 64

Functional Programming methods:

.map()

.filter()

.reduce()

.includes() This used with an empty array and compares a multi-dimensional array, if elem is in the empty array it returns true, else it returns false. So if array is empty it would fill array and if the elem is repeated in first array it will skip that elem. (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/includes>)

function whatIsInAName(collection, source) {

var srcKeys = Object.keys(source);} used to setup a way to test an object to another object. Collection is the bulk object, and source is the condition that needs to be found to filter out the results of the comparison.

We can also generate an array which contains all the keys stored in an object using the Object.keys() method and passing in an object as the argument. This will return an array with strings representing each property in the object. Again, there will be no specific order to the entries in the array.

Here is a very basic approach to checking an object's "value equality".

function isEquivalent(a, b) {

// Create arrays of property names

var aProps = Object.getOwnPropertyNames(a);

var bProps = Object.getOwnPropertyNames(b);

// If number of properties is different,

// objects are not equivalent

if (aProps.length != bProps.length) {

return false;

}

for (var i = 0; i < aProps.length; i++) {

var propName = aProps[i];

// If values of same property are not equal,

// objects are not equivalent

if (a[propName] !== b[propName]) {

return false;

} }

// If we made it this far, objects

// are considered equivalent

return true;

}

// Outputs: true

console.log(isEquivalent(bobaFett, jangoFett));