Module 3

Web APIs and the `global` object

String & Number Methods

Array Methods

Object Methods and ES6 Modules

Soulutions for JS1 Lesson 2.4 Functions

Exercise 1

Make a function, with an apropriate name, that given one parameter, console logs out the value of that parameter plus some descriptive comment.

Invoke the function with a string as its argument.

```
// -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
// Exercise 1
console.log("Exercise 1");

function myOwnLogger (text) {
    console.log ("This is what I log: " + text);
}

myOwnLogger("The text from below");
```

Exercise 2

a) Make a function, isItOdd() that given a numeric parameter, returns `true` if the number passed in is odd, and false if it's not.

b) Invoke the function with a string as its argument. Console log the result.

c) Expand the function to return NaN if the argument passed in is not a valid number.

d) Invoke the function with a string as its argument. Console log the result.

```
// -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
// Exercise 2
console.log("Exercise 2");

// a
function isItOdd(number) {
    if (number % 2 === 1) {
        return true;
    }
    return false;
    // No need for an else here, though you could have one...
}
console.log(isItOdd(13), isItOdd(42));
// true false > just testing, always a good idea

// b
console.log(isItOdd("Whatever")); // false
```

```
// c
function isItOdd2(number) {
    if (isNaN(number)) {
        return NaN;
    } else if (number % 2 === 1) {
        return true;
    }
    return false;
    // No need for an else here, either...
}
console.log(isItOdd2(13), isItOdd2(42.0), isItOdd2("13"));
// true false true > just testing, again

// d
console.log(isItOdd2("Whatever"), isItOdd2("42 whatevers"), isItOdd2("whatevers 42"));
// NaN NaN NaN
```

Exercise 3

Temperatures can be given in Celsius or Farenheit, depending on where we live in the World.

Mathematically you can calculate one from the other in these ways:

```
°F = °C x 9/5 + 32

°C = °F - 32
     -------
       9/5
```

a) Make a function `toFarenheit()` that given one argument, the temperature in celsius, returns the corresponding temperature in farenheit.

Console.log out the corrsponding degrees in farenheit, for the following degrees in calsius `-40`, `0`, `37` and `100`.

b) Make a function `toCelsius()` that given one argument, the temperature in farenheit, returns the corresponding temperature in celsius.

Console.log out the corrsponding degrees in celsius, for the following degrees in farenheit `-40`, `32`, `98.6` and `212`.

Tip: Mind your parenthesis, and note that the solution to one of these can be found on W3School.

```
// -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
// Exercise 3
console.log("Exercise 3");

// a
function toFarenheit(celsius) {
  return celsius * (9/5) + 32;
}

console.log(toFarenheit(-40));  // -40
console.log(toFarenheit(0));    // 32
console.log(toFarenheit(37));   // 98.60000000000001
console.log(toFarenheit(100));  // 212

// b
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}

console.log(toCelsius(-40));    // -40
console.log(toCelsius(32));     // 0
console.log(toCelsius(98.6));   // 37
console.log(toCelsius(212));    // 100
```

Exercise 4

Given two paragraphs in your HTML, with the ids `emptyParagraph` and `anotherParagraph`:

```
<p id="emptyParagraph"></p>
<p id="anotherParagraph"></p>
```

a) Make a function called `writeToPage()`, that takes one parameter, and writes that value of the parameter at text inside the `p#emptyParagraph` element.

Invoke the function with an appropriate argument.

b) Extend the function from a to take a second parameter, `element`, use the value of that parameter to target any element with that name.

Invole the new/extended function, with an appropriate first argument and "anotherParagraph" as the second argument.

c) [Level 2] If the element given in b doesn't exist, make a console **error**, with the text:
`"The given element, " + element + ", doesn't exist."`

Invole the new/extended function, with an appropriate first argument and "whatever" as the second argument.

```
// -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
// Exercise 4
console.log("Exercise 4");

// a
function writeToPage(text) {
    const target = document.getElementById('emptyParagraph')
    target.innerText = text;
}

let myGreeting = "London Calling by The Clash";
writeToPage(myGreeting);

// b
function writeToPage2(text, element) {
    const target = document.getElementById(element);
    target.innerText = text;
}

let myGreeting2 = "This is not America by David Bowie";
writeToPage2(myGreeting2, "anotherParagraph");
```

```
// c
function writeToPage3(text, element) {
    const target = document.getElementById(element);
    if (target !== null) {
        target.innerText = text;
    } else {
        console.error ("The given element, " + element + ", doesn't exist.");
    }
}

let myGreeting3 = "Is there anybody out there by Pink Floyd";
writeToPage3(myGreeting3, "whatever");

// Note that the `console.error()` doesn't stop the rest of the execution of the script. :-)
```

Exercise 5

Given a button and an empty ordered list in your HTML, like this:

```
<button onclick="listItems()">Press Me, I dare you</button>
<ol id="myList"></ol>
```

And a list of programming languages:

```
let programmingLanguages = [
  "Python", "JavaScript",  "Java", "C#", "C", "C++", "Go", "R", "Swift", "PHP"
];
```

Make a function `listItems()` that sorts the list `programmingLanguages` alfabetically, and list them in individual list items in the `ol#myList` element on the page.

Press the button on the webpage to invoke the function.

Hint: To make a list element it's sufficient here to make something like `"<li>" + value + "</li>"`; or `` `<li>${value}</li>` ``;

```
// -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
// Exercise 5
console.log("Exercise 5");

function listItems() {
    const target = document.querySelector("ol#myList");
    let programmingLanguages = [
        "Python",
        "JavaScript",
        "Java",
        "C#",
        "C",
        "C++",
        "Go",
        "R",
        "Swift",
        "PHP"
    ];

    programmingLanguages.sort();

    for (let language of programmingLanguages) {
        target.innerHTML += "<li>" + language + "</li>";
    }
}
```

Exercise 6

Note: In 3a the result from toFarenheit(37), or another value, may be logged out as `98.60000000000001`. Normally we don't want that many decimal points.

a) Based on `toFarenheit()`, make a new function `toFarenheitPrecise()`, that takes two arguments, the temperature in celsius, and the precision (the numbers of decimal points), that returns a float with the required level of precision.
(Level 2: Make the default value for the precision 2)

Console.log out the corrsponding degrees in farenheit, for the following degrees in calsius `-40`, `0`, `37` and `100`.

b) Do the same based on `toCelsius()`, in a new `toCelsiusPrecise()` function.

Console.log out the corrsponding degrees in celsius, for the following degrees in farenheit `-40`, `32`, `98.6` and `212`.

Tip: Use Number.toFixed()

```
// -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
// Exercise 6
console.log("Exercise 6");

// a
function toFarenheitPrecise(celsius, precision = 2) {
  return  Number.parseFloat(celsius * (9/5) + 32).toFixed(precision);
}

console.log( toFarenheitPrecise(-40, 1 ));  // -40.0
console.log( toFarenheitPrecise(0, 1) );    // 32.0
console.log( toFarenheitPrecise(37, 1) );   // 98.6
console.log( toFarenheitPrecise(100, 1) );  // 212.0

// b
function toCelsiusPrecise(fahrenheit, precision = 2) {
  return Number.parseFloat((5/9) * (fahrenheit-32)).toFixed(precision);
}
console.log( toCelsiusPrecise(-40, 1) );    // -40.0
console.log( toCelsiusPrecise(32, 1) );     // 0.0
console.log( toCelsiusPrecise(98.6, 1) );   // 37.0
console.log( toCelsiusPrecise(212, 1) );    // 100.0

// -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Global object

A global object is an object that always exists in the global scope.

In JavaScript, there's always a global object defined. In a web browser, when scripts create global variables defined with the `var` keyword, they're created as members of the global object. (In Node.js this is not the case.) The global object's interface depends on the execution context in which the script is running. For example:

- In a web browser, any code which the script doesn't specifically start up as a background task has a `Window` as its global object. This is the vast majority of JavaScript code on the Web.
- Code running in a `Worker` ** has a `WorkerGlobalScope` object as its global object.
- Scripts running under Node.js have an object called `global` as their global object.

Source

** Using Web Workers

Window

The `Window` interface represents a window containing a DOM document; the document property points to the DOM document loaded in that window.

A window for a given document can be obtained using the `document.defaultView` property.

A global variable, `window`, representing the window in which the script is running, is exposed to JavaScript code.

The `Window` interface is home to a variety of functions, namespaces, objects, and constructors which are not necessarily directly associated with the concept of a user interface window. However, the `Window` interface is a suitable place to include these items that need to be globally available. Many of these are documented in the JavaScript Reference and the DOM Reference.

In a tabbed browser, each tab is represented by its own `Window` object; the global window seen by JavaScript code running within a given tab always represents the tab in which the code is running. That said, even in a tabbed browser, some properties and methods still apply to the overall window that contains the tab, such as `resizeTo()` and `innerHeight`. Generally, anything that can't reasonably pertain to a tab pertains to the window instead.

Window properties, a few examples

`Window.console`

Returns a reference to the console object which provides access to the browser's debugging console.

`Window.document`

Returns a reference to the document that the window contains.

`Window.innerHeight`

Gets the height of the content area of the browser window including, if rendered, the horizontal scrollbar.

`Window.innerWidth`

Gets the width of the content area of the browser window including, if rendered, the vertical scrollbar.

`Window.location`

Gets/sets the location, or current URL, of the window object.

Window methods, a few examples

Window.alert()

Displays an alert dialog.

Window.close()

Closes the current window.

Window.fetch()

Starts the process of fetching a resource from the network.

Window.scroll()

Scrolls the window to a particular place in the document.

Window.setInterval()

Schedules a function to execute every time a given number of milliseconds elapses.

Window.setTimeout()

Schedules a function to execute in a given amount of time.

Fetching data from a server

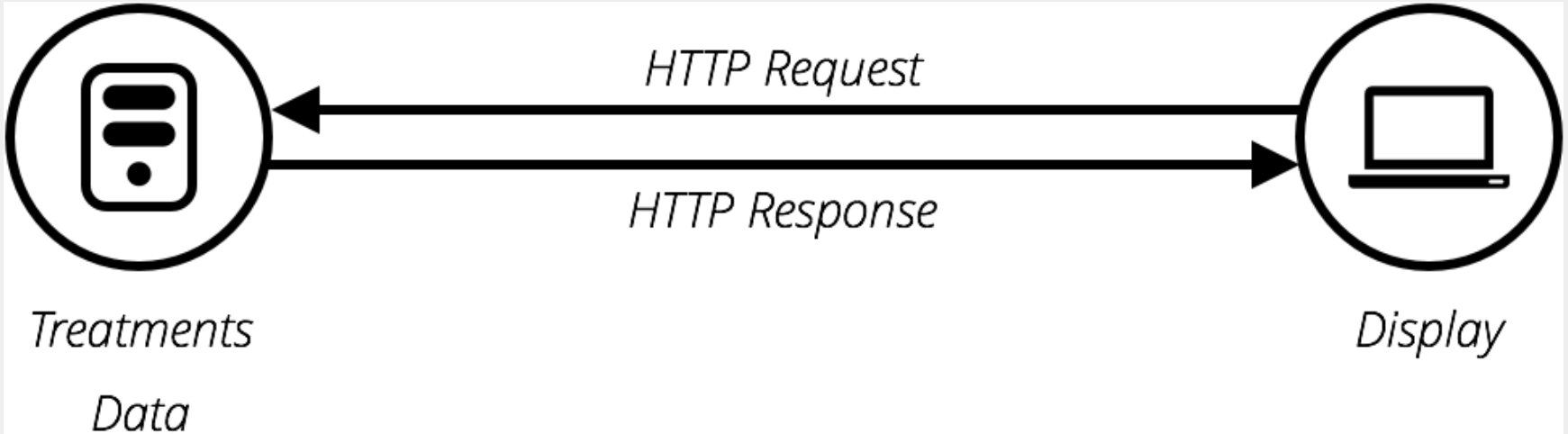
A very common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new page.

This seemingly small detail has had a huge impact on the performance and behavior of sites.

To acheive this we use technologies such as **XMLHttpRequest** and the **Fetch API**.

What is the problem here?

Originally page loading on the web was simple — you'd send a request for a website to a server, and the assets that made the web page would be downloaded and displayed on your computer.



The trouble with this model is that whenever you want to update any part of the page, you've got to load the entire page again.

This is extremely wasteful and results in a poor user experience, especially as pages get larger and more complex.

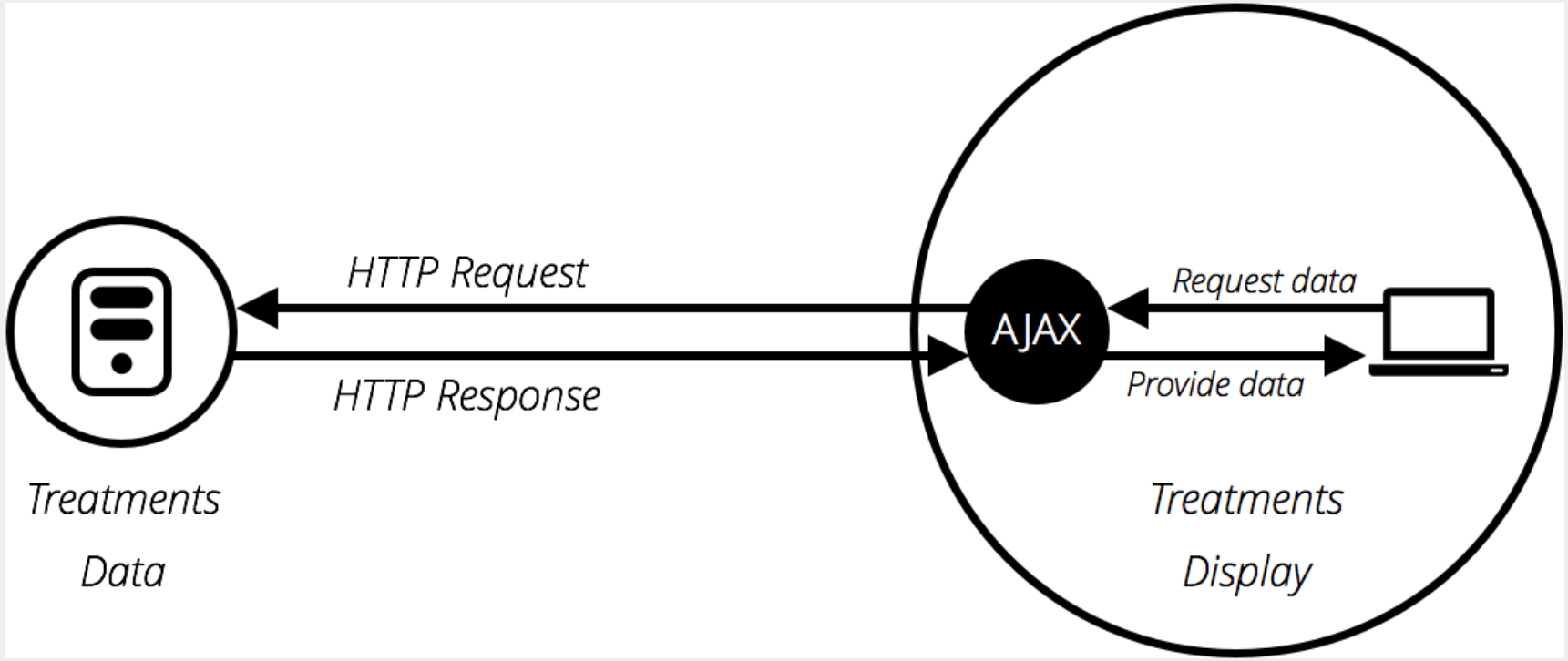
Enter Ajax

This led to the creation of technologies that allow web pages to request small chunks of data (such as HTML, XML, JSON, or plain text) and display them only when needed, helping to solve the problem described above.

Note: In the early days, this general technique was known as **Asynchronous JavaScript and XML** (*Ajax*), because it tended to use XMLHttpRequest to request XML data.

This is normally not the case these days (you'd be more likely to use XMLHttpRequest or Fetch to request JSON), but the result is still the same, and the term "*Ajax*" is still often used to describe the technique.

The Ajax model involves using a web API as a proxy to more intelligently request data rather than just having the browser reload the entire page:

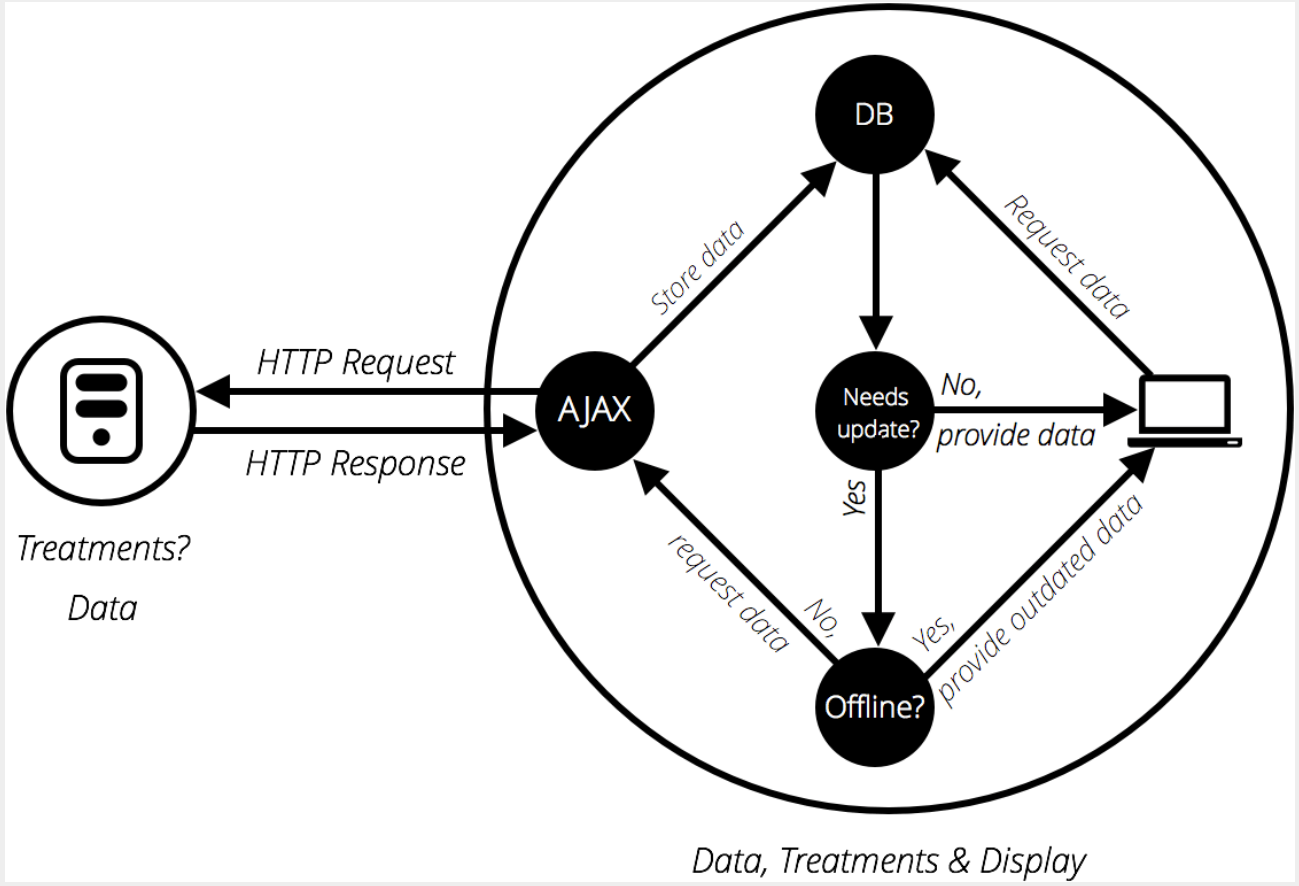


This is a really good thing because:

- Page updates are a lot quicker
- Less data is downloaded on each update

To speed things up even further, some sites also store assets and data on the user's computer when they are first requested, meaning that on subsequent visits they use the local versions instead of downloading fresh copies everytime the page is first loaded.

The content is only reloaded from the server when it has been updated.



A basic Ajax request

Let's look at how such a request is handled, using both **XMLHttpRequest** and **Fetch**.

XMLHttpRequest

XMLHttpRequest (which is frequently abbreviated to **XHR**) is a fairly old technology now — it was invented by Microsoft in the late '90s, and has been standardized across browsers for quite a long time.

To begin creating an XHR request, you need to create a new request object using the `XMLHttpRequest()` constructor. You can call this object anything you like, but we'll call it `request` to keep things simple:

```
let request = new XMLHttpRequest();
```

Next, you need to use the `open()` method to specify what HTTP request method to use to request the resource from the network, and what its URL is. We'll just use the `GET` method here and set an URL as our `url` variable:

```
request.open('GET', url);
request.responseType = 'text'; // Optional
```

Fetching a resource from the network is an asynchronous operation.

The `onreadystatechange` property specifies a function to be executed every time the status of the `XMLHttpRequest` object changes:

```
request.onreadystatechange = function() {...}
```

When readyState property is 4 and the status property is 200, the response is ready:

```
if (this.readyState == 4 && this.status == 200) {
    // Handle request.responseText
}
```

Putting it all together, to get data from a local JSON file, `cats.json` (Download from here):

```
const url = "cats.json";
let xhttp = new XMLHttpRequest();

xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
       let response = xhttp.responseText;
       let myData = JSON.parse(response);
       console.log(myData);
    }
};

xhttp.open("GET", url, true); // true = async
xhttp.send();
```

NOTE: Modern browsers will not run XHR requests if you just run it from a local file. To get around this, you need to test the example by running it through a local web server, like Live Server or MAMP.

Fetch

The **Fetch API** is basically a modern replacement for **XHR**; it was introduced in browsers recently to make asynchronous HTTP requests easier to do in JavaScript.

To convert the XHR example above to Fetch, we make an async function:

```
const url = "cats.json";

async function getData(url) {
    let response = await fetch(url);
    let myData = await response.json();
    console.log(myData);
}

getData(url);
```

Given a HTML file with an `ul` element, with the id `emptyList`:

```
const url = "cats.json";
const out = document.querySelector("ul#emptyList");

async function getData(url) {
    let response = await fetch(url);
    let myData = await response.json();
    console.log(myData);
    displayCats(myData.theStarks, out);
}

getData(url);

function displayCats (list, element) {
    console.log (list, element);
    if (list && element) {
        element.innerHTML = ""; // Clear list
        for (let cat of list) {
            let newCat = `<li>${cat.name} is
                ${cat.gender} and has ${cat.hair}
                fur with ${cat.color}.</li>`;
            element.innerHTML += newCat;
        }
    } else {
        console.error ("Some parameters where wrong: ", list, element);
    }
}
```

Truthy and Falsy

In JavaScript, a truthy value is a value that is considered true when encountered in a Boolean context. All values are truthy unless they are defined as falsy. That is, all values are truthy except `false`, `0`, `-0`, `0n`, `\"\"`, `null`, `undefined`, `NaN`, and `document.all`.

Source: Truthy and Falsy

API

An **application programming interface (API)** is a way for two or more computer programs to communicate with each other. It is a type of software interface, offering a service to other pieces of software.

A document or standard that describes how to build or use such a connection or interface is called an API specification.

A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation.

Whereas a system's user interface dictates how its end-users interact with the system in question, its API dictates how to write code that takes advantage of that system's capabilities.

Examples:

Cat Facts

Use https://cat-fact.herokuapp.com/facts/ to access the Data

Used in the Codepen Fetch (Cat-facts)

Amiibo API

Use https://www.amiiboapi.com/api/amiibo/?name=mario to access data and change parameters to get the data you want. Always check out an APIs Documentation.

Used in the Codepen: XHR (Mario)

Public API Lists

A collective list of free APIs for use in software and web development.

URL: searchParams property

The `searchParams` read-only property of the `URL` interface returns a `URLSearchParams` object allowing access to the `GET` decoded query arguments contained in the URL.

We can use this to:

- retrieving parameters from the query string.
- passing variables to other pages in the query string.

If the URL of your page is `https://example.com/?name=Jonathan%20Smith&age=18` you could parse out the name and age parameters using:

```
let params = new URL(document.location).searchParams;
let name = params.get("name"); // is the string "Jonathan Smith".
let age = parseInt(params.get("age")); // is the number 18
```

Example

See `search-params.html`

HTML:

```
<ul>
    <li><a href="?">No parameters</a></li>
    <li><a href="?id=42">One parameter: id</a></li>
    <li><a href="?id=42&name=Marvin">One parameter: id and name</a></li>
    <li><a href="?q=my+search">One parameter: q</a></li>
    <li><a href="?test=blåbærsyltetøy">One parameter: test</a></li>
    <li><a href="?x=42&y=13&z=55">3 parameters: x, y and z</a></li>
</ul>
```

JavaScript:

```
// get the query string
const queryString = document.location.search;
// create an object that will allows us to access all the query string parameters
const mySearchParams = new URLSearchParams(queryString);

// List all query string parameters
for (const [key, value] of mySearchParams) {
    console.log(key, value);
}

// List value of id
console.log("id has the value: " + mySearchParams.get("id"));

// Note: All params are strings!
if (mySearchParams.get("x") && mySearchParams.get("y") && mySearchParams.get("z")) {
    x = mySearchParams.get("x");
    y = mySearchParams.get("y");
    z = mySearchParams.get("z");
    console.log ("x + y + z = " + (x + y + z));
    console.log ("x + y + z = " + (Number(x) + Number(y) + Number(z)));
}
```

Amiibo-Demo

Todos

Mollify

Read The Global and Window Objects and look at the Lesson Task.