Please note that all the reviewed methods are but a few examples. There are more methods which are not provided in this cheat sheet, and will probably be added as we go.

**Array Methods**

let arr = array.forEach((element, index(optional), array(optional)) => {

// function to execute on each element

});

the **.forEach** executes a provided function once for each array element. It does not return any modified array or a new array.

    codes.forEach((code) => {

        let newOption = `<option value="${code}">${code}</option>`;

        fromSelector.innerHTML += newOption;

        toSelector.innerHTML += newOption;

    })

let arr = array.map((element, index(optional), array(optional)) => {

// return value for new array

return transformedElement;

});

the **.map** provides a new array populated with the results of calling a provided function on every element in the calling array.

    let newCodes = codes.map((code) => `<option>${code}</option>`);

    fromSelector.innerHTML = newCodes.join("");

    toSelector.innerHTML = newCodes.join("");

    console.log(newCodes);

let arr = array.filter((element, index(optional), array(optional)) => {

// return true to keep element, false to discard

return condition;

});

the **.filter** creates a new array with all elements that pass the test implemented by the provided function.

let arr = array.find((element, index(optional), array(optional)) => {

// return true if element satisfies condition

return condition;

});

the **.find** returns the first element in the array that satisfies the provided testing function.

let arr = array.reduce((accumulator, element, index, array) => {

*// return updated accumulator value*

      return updatedAccumulator;

      }, initialValue);

the **.reduce** applies a function against an accumulator and each element in the array from left to right, to reduce it to a single value.

The accumulator – the accumulated value previously returned in the last invocation of the callback or initialValue if supplied.

let arr = array.every((element, index, array) => {

*// return true if element satisfies condition*

       return condition;

       });

the **.every** checks if all elements in the array pass the test implemented by thee provided function.

let arr = array.some((element, index, array) => {

*// return true if element satisfies condition*

       return condition;

       });

the **.some** checks if at least one element in the array satisfies the provided testing function.

**Object Methods**

const obj = { a: 1, b: 2, c: 3 };

const keys = Object.keys(obj);

console.log(keys); *// ['a', 'b', 'c']*

**Object.keys(obj)** returns an **array** of a given object’s own enumerable **property names**.

const obj = { a: 1, b: 2, c: 3 };

const values = Object.values(obj);

console.log(values); *// [1, 2, 3]*

**Object.values(obj)** returns an **array** of a given object’s own enumerable **property values.**

let currency = Object.keys(countryInfo.currencies).join(", ");

    console.log(currency);

let symbol = Object.values(countryInfo.currencies).map(currency => currency.symbol)

    console.log(symbol);

    divP.innerHTML += `<b>Currency:</b> ${currency} - ${symbol}.<br><br>`;

In this example from the countries search project, we ask for the object values of the currencies which is given as an array and then we run the map to find what we need – symbol of the currency.

const obj = { a: 1, b: 2, c: 3 };

const entries = Object.entries(obj);

console.log(entries); *// [['a', 1], ['b', 2], ['c', 3]]*

**Object.entries(obj)** returns an **array** of a given object’s own enumerable **key-value** pairs as **arrays.**

const target = { a: 1, b: 2 };

const source = { b: 3, c: 4 };

const mergedObject = Object.assign({}, target, source);

console.log(mergedObject); *// { a: 1, b: 3, c: 4 }*

**Object.assign(target, ...sources)**: Copies the values of all enumerable own properties from **one or more source objects to a target object**, and returns the **target object**.

The same concept as .assign in modern days is achieved by using the **Spread** (…)

Here are a few examples to the usage of **. . . Spread**

*//Convering string to array of characters*

        let str = 'hello';

        let charArray = [...str];

        console.log(charArray); *// Output: ['h', 'e', 'l', 'l', 'o']*

*//Merging objects*

        let obj1 = { foo: 'bar', x: 42 };

        let obj2 = { hello: 'world', y: 10 };

        let mergedObj = { ...obj1, ...obj2 };

        console.log(mergedObj); *// Output: { foo: 'bar', x: 42, hello: 'world', y: 10 }*

**Loops**

    for (variable of iterable) {

*// code block to execute*

    }

The **for...of** loop in JavaScript is used to iterate over iterable objects (arrays, strings, maps, sets, etc.) and allows you to loop through the elements of an iterable in a concise and readable way.

**variable** : On each iteration, a variable is assigned the value of the next element in the iterable object.

**iterable**: An object that has iterable properties (like arrays, strings, maps, sets, etc.).

const array = ['a', 'b', 'c'];

for (let element of array) {

console.log(element);

}

*// Output:*

*// 'a'*

*// 'b'*

*// 'c'*

const string = 'hello';

for (let char of string) {

console.log(char);

}

*// Output:*

*// 'h'*

*// 'e'*

*// 'l'*

*// 'l'*

*// 'o'*

**Promise / Then and Catch**

A **Promise** in JavaScript is like a **placeholder for an operation that hasn't finished yet**, but will eventually either succeed or fail. It's a way to handle asynchronous operations in a structured manner.

**How to Get a Promise?**

You typically get a Promise when you perform tasks that take time to complete, such as fetching data from a server, reading a file, or waiting for a timer.

Here’s a simple way to display a promise:

let promise = fetch('https://api.example.com/data');

You use .then() to handle the response when the Promise resolves successfully (i.e., when the HTTP request completes successfully).

promise.then((response) => response.json())

*// Convert the response to JSON format*

Chain another .then() to handle the JSON data retrieved from the response.

.then((data) => {

*// Log the data to the console*

   console.log(data);

   })

Use .catch() to handle any errors that occur during the HTTP request or JSON parsing.

.catch ((e) => {

*// Log the error to the console*

   console.log(e);

   });

**Fetching Data**: fetch('https://api.example.com/data') initiates an HTTP request to fetch data from the specified URL. **It returns a Promise (promise).**

**Handling Response**: .then((response) => { ... }) is used to handle the response from the server when the Promise resolves.

**Parsing JSON**: Inside the first .then(), response.json() converts the response to JSON format. **It returns another Promise.**

**Handling Data**: .then((data) => { ... }) is used to handle the data retrieved from the JSON response.

**Handling Errors**: .catch((e) => { ... }) is used to handle any errors that occur during the fetch request or JSON parsing.

In the end it would look like this all together:

        let promise = fetch('https://api.example.com/data');

        promise.then((response) => response.json())

.then((data) => console.log(data))

.catch((e) => console.log(e));

And now in a function

function getCapital1(country) {

    fetch("https://restcountries.com/v3.1/name/" + country)

        .then((response) => response.json())

        .then((data) => console.log(data[0].capital[0]))

        .catch((e) => console.log(e))

};

Combining 2 fetch then and catch:

let weatherApiKey = "YOUR API key here";

function getWeatherOfCapitalCity(country) {

    fetch(`https://restcountries.com/v3.1/name/${country}`)

        .then((response) => response.json())

        .then((data) => {

            let capital = data[0].capital[0];

            fetch(

                `https://api.openweathermap.org/data/2.5/weather?q=${capital}&appid=${weatherApiKey}&units=metric`

            )

                .then((res) => res.json())

                .then((data) => console.log(data))

                .catch((err) => console.log(err));

        })

        .catch((error) => console.log(error));

}

**Async and Await**

**Async Function**:

* An async function is a special type of function in JavaScript that allows you to write asynchronous code more conveniently.
* It's declared using the **async** keyword before the function definition.
* Inside an async function, you can use the **await** keyword to pause the execution of the function until **a promise is settled (resolved or rejected).**

**Await**:

* The await keyword is used inside an async function to pause the execution of the function until **a Promise is settled.**
* It waits for the Promise to resolve and returns the result.
* If the Promise is **rejected**, it throws an **error** that can be caught using **try/catch.**

*async* function fetchData() {

            let response = await fetch('https://api.example.com/data');

            let data = await response.json();

            return data;

        }

        fetchData();

**return** is used to pass data back from an async function to its caller or to terminate the function early with a value.

**console.log()** is used for logging messages or data to the console for debugging or informational purposes, without affecting the function's return value.

When to use async functions and await versus fetch, then and catch.

**Which One to Use?**

* **Readability**: If code readability is a priority, async/await often leads to cleaner and more understandable code, especially for handling multiple asynchronous operations.
* **Compatibility**: If you need to support older browsers or environments where async/await isn't fully supported, using .then() and .catch() with Promises is a safer choice.
* **Preference**: Ultimately, the choice between async/await and .then()/.catch() depends on your team's coding style and the specific requirements of your project. Both approaches are valid and widely used in modern JavaScript development.

**Best Practice:**

* **Mixing Both**: In many cases, developers use a mix of async/await for individual asynchronous functions and .then()/.catch() for handling chains of asynchronous operations, depending on what makes the code clearer and more maintainable.

Combination of the two:

*async* function fetchRandomUser() {

     try {

*// Using async/await with fetch to make an API call*

       let response = await fetch('https://randomuser.me/api/');

*// Parsing JSON data from the response*

       let data = await response.json();

*// Returning the first user object from the results array*

       return data.results[0];

     } catch (error) {

*// Catching any errors that occurred during the fetch operation*

       console.error('Error fetching user:', error);

       }

     };

*// Calling the async function and using .then() and .catch() for handling results*

fetchRandomUser()

     .then(user => {

         console.log('Random User:', user);

*// Further processing or displaying the user data*

         })

     .catch(error => {

         console.error('Error in fetchRandomUser:', error);

*// Handling errors that occurred during the async operation*

         });