**CSS**

|  |  |
| --- | --- |
| p, .blue, #first, div span{ **color** : blue }  This rule applies to:  **<p>**  elements of the blue class  element with the ID first  every **<span>** inside of a **<div>** | div, p { **color**: blue }  So the blue color applies to all **<div>** elements and all **<p>** elements. Without the comma only **<p>** elements that are  a child of a **<div>** would be red. |

**Selector Description**

\* Universal selector (all elements)

Div Tag selector (all **<div>** elements)

.blue Class selector (all elements with class blue)

.blue.red All elements with class blue **and** red (a type of Compound selector)

#headline ID selector (the element with "id" attribute set to headline)

:pseudo-class All elements with pseudo-class

::pseudo-element Element that matches pseudo-element

:lang(en) Element that matches :lang declaration, for example **<span** lang="en"**>**

div > p child selector  
  
**Overview**

**Selector Description**

div span Descendant selector (all **<span>**s that are descendants of a **<div>**)

div > span Child selector (all **<span>**s that are **a direct child** of a **<div>**)

a ~ span General Sibling selector (all **<span>**s that are siblings after an **<a>**)

a + span Adjacent Sibling selector (all **<span>**s that **are immediately after an** **<a>**)

input:not([disabled]):not(.example){

**background-color**: #ccc;

}

input:not([disabled], .example){

**background-color**: #ccc;

}

Hex code is used to denote RGB components of a color in base-16 hexadecimal notation. #ff0000,

body {

**background-color**: #de1205; */\* red \*/*

}

**RGB / RGBa**

RGB stands for Red, Green and Blue, and requires of three separate values between 0 and 255,

RGBa allows you to add an additional alpha parameter between 0.0 and 1.0 to define opacity.

header {

**background-color**: rgb(0, 0, 0); */\* black \*/*

}

footer {

**background-color**: rgba(0, 0, 0, 0.5); */\* black with 50% opacity \*/*

}

**HSL / HSLa**

HSL stands for hue, saturation, and lightness, and is also often called HLS:

Hue is a degree on the color wheel (from 0 to 360).

Saturation is a percentage between 0% and 100%.

Lightness is also a percentage between 0% and 100%.

li a {

**background-color**: hsl(120, 100%, 50%); */\* green \*/*

}

#p1 {

**background-color**: hsla(120, 100%, 50%, .3); */\* green with 30% opacity \*/*

}

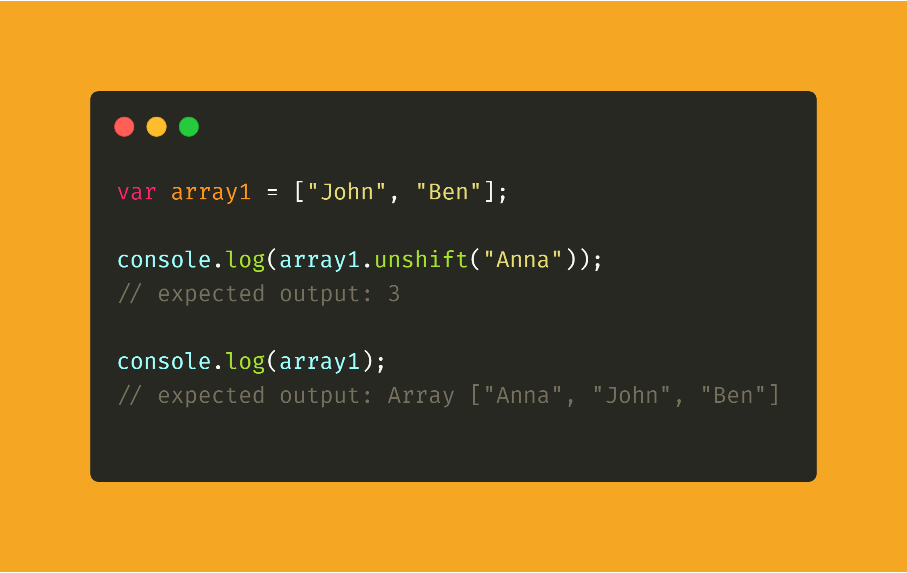
**JS**

**Array Methods:**

|  |  |
| --- | --- |
| **Array.from()**  Array.from() lets you create Arrays from:   * array-like objects (objects with a length property and indexed elements); or * [iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable) (objects such as [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) and [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)). | Array.from('foo');  // [ "f", "o", "o" ] |
| **Array.isArray()**  If the value is an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), true is returned; otherwise, false is. | Array.isArray([1, 2, 3]); // true  Array.isArray({foo: 123}); // false  Array.isArray('foobar'); // false  Array.isArray(undefined); // false |
| **Array.of()**  method creates a new Array instance from a variable number of arguments, regardless of number or type of the arguments. | Array.of(7); // [7]  Array(7); // array of 7 empty slots  Array.of(1, 2, 3); // [1, 2, 3]  Array(1, 2, 3); // [1, 2, 3] |
| **concat()** method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array. | const array1 = ['a', 'b', 'c'];  const array2 = ['d', 'e', 'f'];  const array3 = array1.concat(array2);  console.log(array3);  // expected output: Array ["a", "b", "c", "d", "e", "f"] |
| **copyWithin()**  method shallow copies part of an array to another location in the same array and returns it without modifying its length. | arr.copyWithin(target[, start[, end]]) const array1 = ['a', 'b', 'c', 'd', 'e'];  // copy to index 0 the element at index 3  console.log(array1.copyWithin(0, 3, 4));  // expected output: Array ["d", "b", "c", "d", "e"] |
| **entries()** method returns a new **Array Iterator** object that contains the key/value pairs for each index in the array | const a = ['a', 'b', 'c'];  for (const [index, element] of a.entries())  console.log(index, element);  // 0 'a'  // 1 'b'  // 2 'c' |
| **every()**   method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value | const isBelowThreshold = (currentValue) => currentValue < 40;  const array1 = [1, 30, 39, 29, 10, 13];  console.log(array1.every(isBelowThreshold));  // expected output: true |
| **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. | const animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];  console.log(animals.slice(2));  // expected output: Array ["camel", "duck", "elephant"]  console.log(animals.slice(2, 4));  // expected output: Array ["camel", "duck"]  console.log(animals.slice(1, 5));  // expected output: Array ["bison", "camel", "duck", "elephant"] |

|  |  |
| --- | --- |
| **splice()**  method changes the contents of an array by removing or replacing existing elements and/or adding new elements [in place](https://en.wikipedia.org/wiki/In-place_algorithm). | const months = ['Jan', 'March', 'April', 'June'];  months.splice(1, 0, 'Feb');  // inserts at index 1  console.log(months);  // expected output: Array ["Jan", "Feb", "March", "April", "June"]  months.splice(4, 1, 'May' );  // replaces 1 element at index 4  console.log(months);  // expected output: Array ["Jan", "Feb", "March", "April", "May"]  const myFish = ["angel", "clown", "drum", "sturgeon"];  const removed = myFish.splice(2, 1, "trumpet");  // myFish is ["angel", "clown", "trumpet", "sturgeon"]  // removed is ["drum"] |

Push & pop  


Shift & unshift:  


**Currying:**fn takes multiple arguments & turning it to a sequence of fns each with single argument.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | const customMiddleWare =store=>next=>action=> { | |  | console.log("Middleware triggered:", action); | |  | next(action); | |  | } |   function log(date, importance, message) {  alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);  }  log = \_.curry(log);   log(new Date())("DEBUG")("some debug"); // log(a)(b)(c) // logNow will be the partial of log with fixed first argument  let logNow = log(new Date());  // use it  logNow("INFO", "message"); // [HH:mm] INFO message let debugNow = logNow("DEBUG");  debugNow("message"); // [HH:mm] DEBUG message   function curry(func) {  return function curried(...args) {  if (args.length >= func.length) {  return func.apply(this, args);  } else {  return function(...args2) {  return curried.apply(this, args.concat(args2));  }  }  };  } |

**Recurssion:**Recursion is a process of calling itself.

|  |
| --- |
| function factorial(x)  {  if (x === 0)  {  return 1;  }  return x \* factorial(x-1);    } |

**IIFE:**

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

|  |
| --- |
| (function () { statements })(); |

It is a design pattern which is also known as a [Self-Executing Anonymous Function](https://developer.mozilla.org/en-US/docs/Glossary/Self-Executing_Anonymous_Function) and contains two major parts:

1. The first is the anonymous function with lexical scope enclosed within the [Grouping Operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Grouping) (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
2. The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.

var result = (function () {  
 var name = "Barry";  
 return name;})();// Immediately creates the output:

result; // "Barry"

**Closures:**The inner function can access the variables of the enclosing function due to closures in JavaScript. In other words, the inner function preserves the scope chain of the enclosing function at the time the enclosing function was executed, and thus can access the enclosing function’s variables.  
  
function outer() {var b = 10; function inner() {  var a = 20;  console.log(a+b); } return inner;}var X = outer(); **console.dir(X); //use console.dir() instead of console.log()**



function outer() {var b = 10;**var c = 100;** function inner() {  var a = 20;   **console.log("a= " + a + " b= " + b);**

**a++;  
 b++;** } return inner;}var X = outer(); // outer() invoked the first time  
var Y = outer(); // outer() invoked the second time  
//end of outer() function executions  
X(); // X() invoked the first time  
X(); // X() invoked the second time  
X(); // X() invoked the third time

Y(); // Y() invoked the first time

Console:

a=20 b=10  
a=20 b=11  
a=20 b=12  
a=20 b=10

the value of outer function is stored in the inner function and doesn’t initiate it again.

**Callback:**

**Simply put:** A callback is a function that is to be executed **after** another function has finished executing — hence the name ‘call back’.

**More complexly put:** In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions. Functions that do this are called **higher-order functions**. Any function that is passed as an argument is called a **callback function**.

Why do we need Callbacks?

For one very important reason — JavaScript is an event driven language. This means that instead of waiting for a response before moving on, JavaScript will keep executing while listening for other events

|  |
| --- |
| function doHomework(subject, callback) {  alert(`Starting my ${subject} homework.`);  callback();  }  doHomework('math', function()  {alert('Finished my homework');  }); |

**Higher Order Function:**

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

Higher-order functions allow us to abstract over actions, not just values.

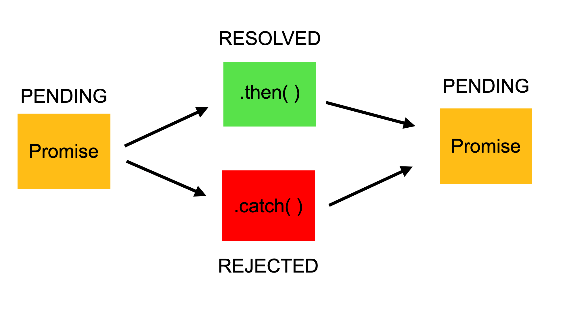
Probably the greatest benefit of HOFs is greater reusability.

|  |
| --- |
| totalAge = users.reduce((total, user) => user.age + total, 0);  namesStartingWithB = users.filter((user) => startsWithB(user.name)); |

**Promise:**

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred)

* **Pending:** Initial State, before the Promise succeeds or fails
* **Resolved:** Completed Promise
* **Rejected:** Failed Promise



### What is the difference between Callbacks and Promises?

The main difference between Callback Functions and Promises is that we attach a callback to a Promise rather than passing it. So we still use callback functions with Promises, but in a different way (chaining).

[Callback hell](http://callbackhell.com/):

Without promise – passing the callback fn

firstRequest(function(response) {

secondRequest(response, function(nextResponse) {

thirdRequest(nextResponse, function(finalResponse) {

console.log('Final response: ' + finalResponse);

}, failureCallback);

}, failureCallback);

}, failureCallback);

//Using Promise – attaching the callback function

firstRequest()

.then(function(response) {

return secondRequest(response);

}).then(function(nextResponse) {

return thirdRequest(nextResponse);

}).then(function(finalResponse) {

console.log('Final response: ' + finalResponse);

}).catch(failureCallback);

.

|  |
| --- |
| const myPromise = new Promise((resolve, reject) => {  let condition;  if(condition is met) {  resolve('Promise is resolved successfully.');  } else {  reject('Promise is rejected');  }  }); |

**Object oriented Js:**

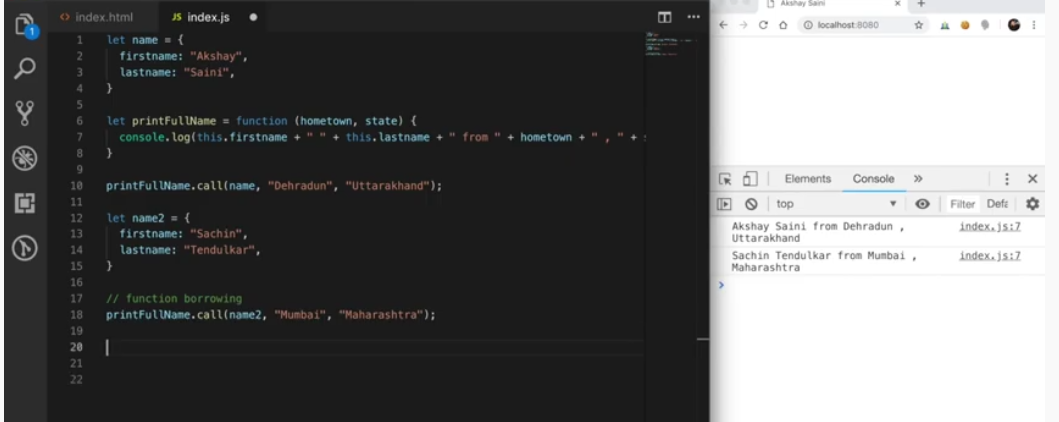
**Inheritance:** objects can inherit features from other objects. ----- class (extend keyword)  
 **Polymorphism**: objects can share the same interface—how they are accessed and used—while their underlying implementation of the interface may differ.------ override any prop in parent class

Child class method will override the parent clas method if the method name is same.

**Encapsulation**: each object is responsible for specific tasks.----- Closures

**Apply/ call/ bind:**

**Call:**

To call the method with the given this obj as the input reference.

Fnname.Call(this reference obj, other inputs params)

|  |
| --- |
| var employee1 = { firstName: "John", lastName: "Rodson" };  var employee2 = { firstName: "Jimmy", lastName: "Baily" };  function invite(greeting1, greeting2) {  console.log(  greeting1 + " " + this.firstName + " " + this.lastName + ", " + greeting2  );  }  invite.call(employee1, "Hello", "How are you?"); // Hello John Rodson, How are you?  invite.call(employee2, "Hello", "How are you?"); // Hello Jimmy Baily, How are you? |

**Apply:**  
  
Same as call , inputs are passed in array here.

Fnname.Call(this reference obj,[param1, param2, param3])

|  |
| --- |
| var employee1 = { firstName: "John", lastName: "Rodson" };  var employee2 = { firstName: "Jimmy", lastName: "Baily" };  function invite(greeting1, greeting2) {  console.log(  greeting1 + " " + this.firstName + " " + this.lastName + ", " + greeting2  );  }  invite.apply(employee1, ["Hello", "How are you?"]); // Hello John Rodson, How are you?  invite.apply(employee2, ["Hello", "How are you?"]); // Hello Jimmy Baily, How are you? |

**Bind:**

Bind creates a new function that will have this set to the first parameter passed to bind().

Returnsthe method itself to the assigned variable instead of the result as like call/apply.

Let newfn = fn.bind(thisreference, param1, param2, param3);

In constructor:

this.handleClick = this.handleClick.bind(this);

|  |
| --- |
| var employee1 = { firstName: "John", lastName: "Rodson" };  var employee2 = { firstName: "Jimmy", lastName: "Baily" };  function invite(greeting1, greeting2) {  console.log(  greeting1 + " " + this.firstName + " " + this.lastName + ", " + greeting2  );  }  var inviteEmployee1 = invite.bind(employee1);  var inviteEmployee2 = invite.bind(employee2);  inviteEmployee1("Hello", "How are you?"); // Hello John Rodson, How are you?  inviteEmployee2("Hello", "How are you?"); // Hello Jimmy Baily, How are you? |

**Composition:**

**Function composition** is the process of combining two or more functions to produce a new function. Composing functions together is like snapping together a series of pipes for our data to flow through.

Without composition expamle:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | const toSlug = input => encodeURIComponent( | |  | input.split(' ') | |  | .map(str => str.toLowerCase()) | |  | .join('-') | |  | ); | |

Compose multiple function: const compose = (...fns) => x => fns.reduceRight((v, f) => f(v), x);

Reduceright-> iterate it from right to left.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | const toSlug = compose( | |  | encodeURIComponent, | |  | join('-'), | |  | map(toLowerCase), | |  | split(' ') | |  | ); | |  |  | |  | console.log(toSlug('JS Cheerleader')); | |

Available in lodash too , import { compose } from 'lodash/fp';

Recursion:

To check if an value is present in the nested obj

var nestedObject = {

    data: {

        info: {

            stuff: {

                thing: {

                    moreStuff: {

                        magicNumber: 44,

                        something: 'foo2'

                    }

                }

            }

        }

    }

}

let hasIt = contains(nestedObject, 44); // true

let doesntHaveIt = contains(nestedObject, "foo"); // false

console.log(hasIt, doesntHaveIt);

function contains(obj, value){

    for(var key in obj){

        if(typeof obj[key] === 'object'){

            return contains(obj[key], value);

}

        if (obj[key] === value){

            return true;

        }

    }

    return false;

}

Flatening using reduce:

const data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

const flat = data.reduce((total, amount) => {

return total.concat(amount);

}, []);

flat // [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

React js:

## **Custom Attributes**

Custom attributes allow you to attach other values onto an HTML element.

Custom attributes will always start with data-.

<div className="cart-item">

<span className="item-name">{item.name}</span>

<button onClick={this.removeItem} data-remove={item.id}>

x

</button>

</div>

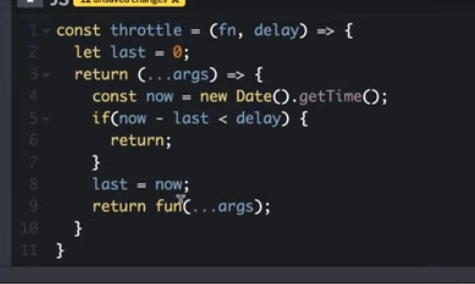
// ...

removeItem = e => {

const removeId = e.target.dataset.remove;

data- value is available in e.target.dataset

Throttle:



Debounce:

|  |  |
| --- | --- |
|  | function debounce(cb, delay = 250) {  let timeout  return (...args) => {  clearTimeout(timeout)  timeout = setTimeout(() => {  cb(...args)  }, delay)  }  } |

const countOccurrences = (arr, val) => arr.reduce((a, v) => (v === val ? a + 1 : a), 0);

console.log(countOccurrences([1, 1, 2, 1, 2, 3], 1));

**ErrorBoundary**

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

// Update state so the next render will show the fallback UI.

return { hasError: true };

}

componentDidCatch(error, errorInfo) {

// You can also log the error to an error reporting service

logErrorToMyService(error, errorInfo);

}

render() {

if (this.state.hasError) {

// You can render any custom fallback UI

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

**Router**

Browser Router: build based on browser’s History API, push/back (window.hisory)

Hash Router: get the path after the # in the url

Memory Router: same as browser router the history is stored in memory for other than browser environment (mobile/ automation testing)

### possible ways to create objects in JavaScript

1.) var object = new Object();

2.) var object = Object.create(null);

3.) var object = {

name: "Sudheer",

age: 34

};

4.). function Person(name) {

this.name = name;

this.age = 21;

}

5.) var object = new Person("Sudheer");

6.) var object = new (function () {

this.name = "Sudheer";

})();

**Prototype chaining** is used to build new types of objects based on existing ones. It is similar to inheritance in a class based language.

|  |
| --- |
| const memoizAddition = () => {  let cache = {};  return (value) => {  if (value in cache) {  console.log("Fetching from cache");  return cache[value]; // Here, cache.value cannot be used as property name starts with the number which is not a valid JavaScript identifier. Hence, can only be accessed using the square bracket notation.  } else {  console.log("Calculating result");  let result = value + 20;  cache[value] = result;  return result;  }  };  };  // returned function from memoizAddition  const addition = memoizAddition();  console.log(addition(20)); //output: 40 calculated  console.log(addition(20)); //output: 40 cached |