**Topics covered:**

1. Block Scope.- done

2. Global Scope. - done

3. Lexical Scope. - done

4. Data type. -done

5. Math Library.

6. Number Library.

7. Strings.

8. Prototype inheritance - done

9. Boolean.

10. Symbols

11. Type conversion - done

12. Operators.

13. Statements.

14. Reference Data type. (Class, Object, array, etc) - done

15. How to add methods to object. -done

16. Maps and Sets.

17. Function declaration & expression - done

18. Default parameter.

19. Spread operators.

20. Call back Function - done

21. OOP's in JS.

22. Arrow Functions. – half done

23. Functional programming. Call, apply & bind. - done

24. Closures. - done

25. Asynchronous in JS - done

26. Promises - done

27. DOM & navigation.

28. Event Bubbling - done

29. Proxy reflection.

30. Regex

31. Web Workers

32. Generators.

33. Debouncing - done

34. Throttling - done

35. Higher Order Functions - done

36. Recursive - done

37. Currying - done

38. Polyfills for bind – done

39. Delegation in Javascript – done

40. Async vs Differ – done

41. Service workers

npx http-server

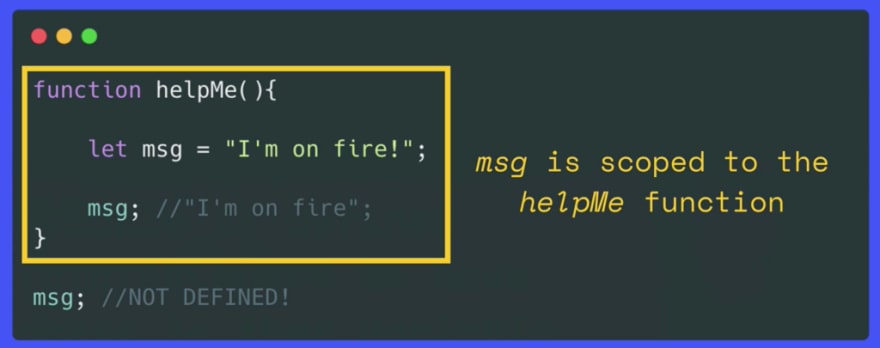
**Function, Block and Lexical Scope. What's the difference?**

**What is Scope?**

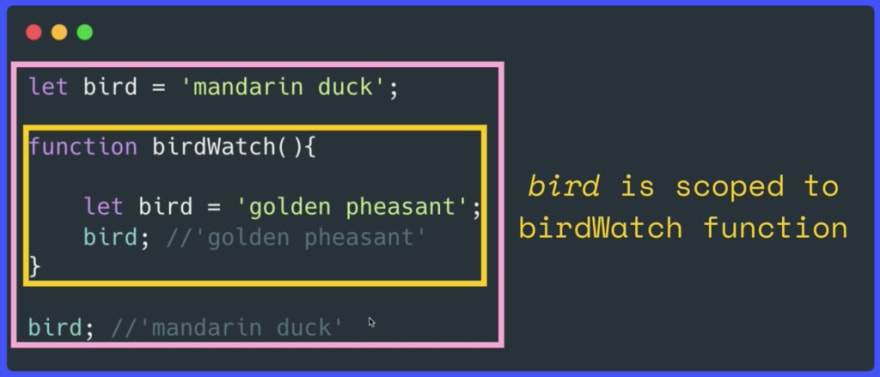
Scope is the location where a variable is defined and the context where other pieces of your code can access and manipulate it. In this post I'll outline the three different types of scopes and how they behave in JavaScript.

**Function Scope**

In the below example a function called helpMe has a variable called msg. Msg is **function scoped** meaning that if you call it outside the helpMe function you'll get undefined.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--O2zXrF05--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/codingnotesbyella.files.wordpress.com/2020/10/screen-shot-2020-10-15-at-16.30.37.png%3Fw%3D1024)

If you have two variables with the same name their location matters. In the below example a bird variable is declared outside of a function (pink box) and then a bird variable is function scoped to birdWatch. When bird is outside the function the 'mandarin duck' is printed.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--e6MZz0gp--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/codingnotesbyella.files.wordpress.com/2020/10/screen-shot-2020-10-18-at-16.44.02.png%3Fw%3D1024)

**Block Scope**

Blocks of code are defined with curly braces. It's important to note that Let and Const are **block scoped.**This means when you declare a variable using Let or Const those variables can't be accessed outside that block.

**Example 1:**

In the following example a is stored in global scope. B and c are stored in bloke scope hence the name blocked scoped .

{

Var a = 10;

let b = 4;

const c = 5;

}

Console.log(a); // Output : 10

Console.log(b); // Output : undefined

Console.log(c); // Output : undefined

**Example 2:**

In the following example b outside the scope(“{}”) is stored in script scope. That’s why when then b inside block scope in changed it doesn’t change the value of b. When we try to access the value of be outside the scope we get 100 which is from the script scope.

Let b = 100;

{

Var a = 10;

let b = 4;

const c = 5;

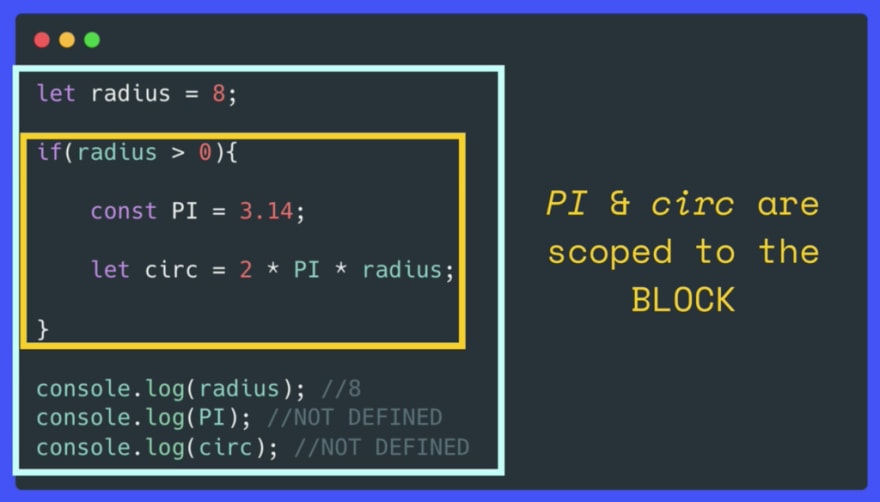
Console.log(a); // Output : 10

Console.log(b); // Output : 4

Console.log(c); // Output : 5

}

Console.log(c); // Output : 100

[](https://res.cloudinary.com/practicaldev/image/fetch/s--tBV-3bee--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/codingnotesbyella.files.wordpress.com/2020/10/screen-shot-2020-10-15-at-16.27.02.png%3Fw%3D1024)

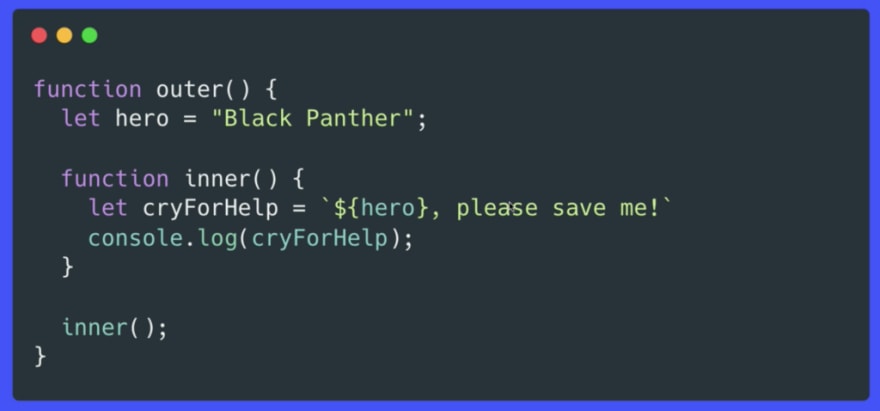
In contrast, if you use **var** you can still access a variable that is declared inside a block scope.

// let & const are BLOCK SCOPED  
  
if (true) {  
const animal = 'eel';  
console.log(animal); //'eel'  
}  
console.log(animal); //NOT DEFINED!  
  
// Var is not BLOCK SCOPES  
if (true) {  
var animal = 'eel';  
console.log(animal); //'eel'  
}  
console.log(animal); //'eel'

**Lexical Scope**

Lexical scope refers to the fact that nested functions are scoped to their **parent/outer functions** (but it's a one way relationship).

In the below example the inner function is nested inside the outer function. As you can see we have access to hero inside the inner function. But outside this function we do not have access to inner.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--RXib_8l0--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/codingnotesbyella.files.wordpress.com/2020/10/screen-shot-2020-10-15-at-16.27.38.png%3Fw%3D1024)

Below is another example of how lexical scope works. When we call outer() the below example prints out 'AMADEUS'. This is because movie isn't defined inside the extraInner function so the function looks for the nearest movie which is Amadeus. If The Shining wasn't commented out the console would print it.

function outer() {  
 let movie = 'Amadeus';  
  
 function inner() {  
 // let movie = "The Shining";   
  
 function extraInner() {   
 //movie is not defined in this function   
 //but it has access to parent function's variables  
 console.log(movie.toUpperCase())   
 }   
 extraInner();  
 }  
 inner();  
}  
  
outer(); //'AMADEUS'

/**/Order of how the function is called**  
  
2. function outer() {  
 3. let movie = 'Amadeus';  
  
 5. function inner() {  
 6. let movie = "The Shining";  
  
 8. function extraInner() {  
 9. console.log(movie.toUpperCase())  
 }  
 7. extraInner();  
 }  
4. inner();  
}  
   
1. outer();  
   
10. >>>'THE SHINING'

**Reference Url**: https://dev.to/ellawilksharper/function-block-and-lexical-scope-what-s-the-difference-575k/

**How to add methods to object**

**Example 1:**

function Foo () { var secret = 38; this.name = "Bob"; }

Foo.prototype.bar = function () { console.log(secret); };

Foo.prototype.otherFunc = function () { console.log(this.name); };

var myFoo = new Foo();

myFoo.otherFunc(); // "Bob";

myFoo.bar(); // error -- `secret` is undefined...

// ...or a value of `secret` in a higher/global scope

**Example 2:**

Const person = {

firstName : “Anish”,

lastName : “Shrestha”,

getFullName : function(){

return this.firstName + this.lastName;

}

}

**Example 3:**

Var obj = new Object();

obj.firstName = “Anish”;

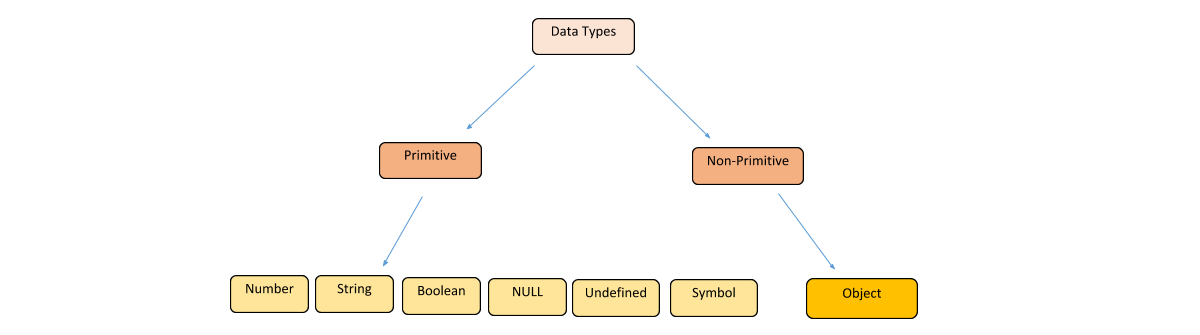
obj.lastName = “Shrestha”;

obj.getFullName = function(){

return this.firstName + this.lastName ;

}

**Different Data Types in JavaScript**



Reference link: <https://www.edureka.co/blog/data-types-in-javascript/>

Primitive data types are not mutable.

**Type Conversion:**

Ref URL: https://www.w3schools.com/js/js\_type\_conversion.asp

**Functions:**

**Function Declaration:**

Function xyz () {

}

**Function Expression:**

Var b = function () (

}

**Named functional expression:**

Var b = function xyz () {

}

But xyz cannot be called here like xyz() it will give undefined error.

**First class functions:**

The ability to use functions as values is first class functions

**Example:** Functions can be passed as an arguments and function can be returned from the functions.

**Arrow function**:

Only included in ES6

Var getName = () => {

Return “Anish Shrestha”

}

**Difference between undefined and not defined:**

Console.log(x)

In this case x is not defined

Var x;

In this case x is undefined

**Closure:**

A closure is the combination of a function and the lexical environment within which that function was declared. Closure is a function which is bind with its lexical environment. A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

**Closure Scope Chain**

Every closure has three scopes:

* Local Scope (Own scope)
* Outer Functions Scope
* Global Scope

**Example 1:**

// global scope

var e = 10;

function sum(a){

return function(b){

return function(c){

// outer functions scope

return function(d){

// local scope

return a + b + c + d + e;

}

}

}

}

console.log(sum(1)(2)(3)(4)); // log 20

// You can also write without anonymous functions:

// global scope

var e = 10;

function sum(a){

return function sum2(b){

return function sum3(c){

// outer functions scope

return function sum4(d){

// local scope

return a + b + c + d + e;

}

}

}

}

var sum2 = sum(1);

var sum3 = sum2(2);

var sum4 = sum3(3);

var result = sum4(4);

console.log(result) //log 20

**Example2:**

Function a (){

Var a = 10;

Function b () {

Console.log(a);

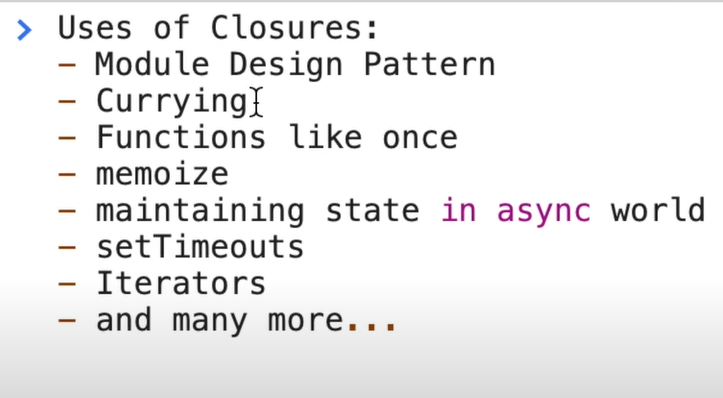
}

b();

}

a();

Ref URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures



**What is Module Pattern**

The Module Pattern is one of the important patterns in JavaScript. It is a commonly used Design Pattern which is used to wrap a set of variables and functions together in a single scope.

It is used to define objects and specify the variables and the functions that can be accessed from outside the scope of the function.

We expose certain properties and function as public and can also restrict the scope of properties and functions within the object itself, making them private.

This means that those variables cannot be accessed outside the scope of the function.

We can achieve data hiding an abstraction using this pattern

Ref URL: <https://coryrylan.com/blog/javascript-module-pattern-basics#:~:text=The%20Module%20Pattern%20is%20one,for%20services%20and%20testing%2FTDD>.

**Inheritance and the prototype chain**

Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this prototype chain.

**Higher Order Function:**

A function which takes another function as an argument or returns a function is called higher order function. It is used in modular programming.

**Functional Programming:**

Thinking the logic in terms of smaller functions and passing them into another function, treating function as values

**Call, Apply and Bind:**

**Call**:

Used for calling function which binds its first argument as this in the function called.

**Example1:**

Let name = {

firstName: “Anish”,

lastName : “Shrestha”,

}

Let getFullName = function(){

return this.firstName + “ ” + this.secondName // here this is referred to name

}

// so to call the method getfullName

getFullName.call(name);

Let Name1 = {

firstName: “Manjushree”,

lastName : “Shrestha”,

}

// Now we need to call the function

name.getFullName.call(name1) // so name1 will be bind with this when the function is called upon

**Example2:**

If we want to pass an extra argument to the getFullName

getFullName = function(middleName) {

return this.firstName + “ ” + middleName + “ ” + this.lastName

}

We can do this by :

getFullName.call(name, “Bahadur”);

In here the second argument is the extra parameter which we want to send as parameter in the function

Apply:

The only difference between apply and call is how we send the parameters. So in apply we send extra parameters as an array.

**Example:**

Let getFullName = function(middleName, city) {

Console.log(this.firstName + “ ” + middleName + “ ” + this.lastName + “ City: ” + city);

}

getFullName.apply(name,[“Bahadur”, “lalitpur”]);

Bind :

The only difference between bind and call is that bind gives us copy of that method which we can call later rather than calling it in same moment which is done in call.

**Debouncing:** used in amazon/flipkart

Limit rate of function calls. Calls when time difference between two key press events is greater than the limit. Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, that it stalls the performance of the web

Example: Will occur after the resize and waits for 300ms (limit) and then trigger the resize event

Reference URL: https://www.geeksforgeeks.org/debouncing-in-javascript/

**Throttling:**

Function call is made after an interval. Limit rate of function call by call after certain limit of time. The difference between two function call events.

Example: Will occur resize event every 300ms (limit)

**Event Bubbling and Capturing also known as Trickling:**

Example Event Bubbling:

**Html:**

 <div id="grandparent">

      <div id="parent">

        <div id="child"></div>

      </div>

    </div>

Javascript:

Document.querySelector(“grandparent”).addEventListener(‘click’,()=>{

Console.log(“G”);

},false); // false is by a default parameter which is for event bubbling

Document.querySelector(“parent”).addEventListener(‘click’,()=>{

Console.log(“P”);

})

Document.querySelector(“grandparent”).addEventListener(‘click’,()=>{

Console.log(“C”);

})

**Output:**

**C**

**P**

**G**

**Example event capturing:**

For event bubbling we only change the last parameter to true.

**Javascript:**

Document.querySelector(“grandparent”).addEventListener(‘click’,()=>{

Console.log(“G”);

},true); // true is for event capturing

Document.querySelector(“parent”).addEventListener(‘click’,()=>{

Console.log(“P”);

},true);

Document.querySelector(“grandparent”).addEventListener(‘click’,()=>{

Console.log(“C”);

},true);

**Output:**

G

P

C

**Stop Propagation:**

Now to stop from the propagation of the event like from bubbling or capturing. We use event.stopPropagation();

Example:

Document.querySelector(“grandparent”).addEventListener(‘click’,(e)=>{

Console.log(“G”);

e.stopPropagation();

},true);

**Event Delegation:**

Attaching event handler to parent rather than attaching all to its child element. Rather than putting click button in every li element we put click event in a div which is enclosing the li.

**HTML:**

<ul id="categories">

   <li id="laptop"></li>

   <li id="shoes"></li>

   <li id="monitors"></li>

 </ul>

**Javascript:**

document.querySelector("#categories").addEventListener(

  "click",

  (e) => {

    console.log(e.target.id);

  },

  false

);

Advantage:

Suppose we are adding new element in the list while we are scrolling down the page. If we had added event handler to the child itself then we had to attach the event to each child dynamically.

Now, If we attach the event listener to the parent then that issues of attaching event to all child element is resolved.

Click, keyup etc. events are bubbled up.

**Asynchronous Javascript:**

* Callbacks
* Promises
* Async/await

**Webapis: All the webapis can be accessed through global object that is window.**

1. setTimeout //second priority, is stored in callback queue
2. dom Apis: example: document.getelementbyid();
3. fetch() // first priority, is stored in microtask queue
4. localstorages
5. console
6. location

**Differences between Async and differ:**

# Scripts: async, defer

In modern websites, scripts are often “heavier” than HTML: their download size is larger, and processing time is also longer.

When the browser loads HTML and comes across a <script>...</script> tag, it can’t continue building the DOM. It must execute the script right now. The same happens for external scripts <script src="..."></script>: the browser must wait for the script to download, execute the downloaded script, and only then can it process the rest of the page.

That leads to two important issues:

1. Scripts can’t see DOM elements below them, so they can’t add handlers etc.
2. If there’s a bulky script at the top of the page, it “blocks the page”. Users can’t see the page content till it downloads and runs:

<p>...content before script...</p>

<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<!-- This isn't visible until the script loads -->

<p>...content after script...</p>

There are some workarounds to that. For instance, we can put a script at the bottom of the page. Then it can see elements above it, and it doesn’t block the page content from showing:

<body>

...all content is above the script...

<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

</body>

But this solution is far from perfect. For example, the browser notices the script (and can start downloading it) only after it downloaded the full HTML document. For long HTML documents, that may be a noticeable delay.

Such things are invisible for people using very fast connections, but many people in the world still have slow internet speeds and use a far-from-perfect mobile internet connection.

Luckily, there are two <script> attributes that solve the problem for us: defer and async.

## [defer](https://javascript.info/script-async-defer" \l "defer)

The defer attribute tells the browser not to wait for the script. Instead, the browser will continue to process the HTML, build DOM. The script loads “in the background”, and then runs when the DOM is fully built.

Here’s the same example as above, but with defer:

<p>...content before script...</p>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<!-- visible immediately -->

<p>...content after script...</p>

In other words:

* Scripts with defer never block the page.
* Scripts with defer always execute when the DOM is ready (but before DOMContentLoaded event).

The following example demonstrates the second part:

<p>...content before scripts...</p>

<script>

document.addEventListener('DOMContentLoaded', () => alert("DOM ready after defer!"));

</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<p>...content after scripts...</p>

1. The page content shows up immediately.
2. DOMContentLoaded event handler waits for the deferred script. It only triggers when the script is downloaded and executed.

**Deferred scripts keep their relative order, just like regular scripts.**

Let’s say, we have two deferred scripts: the long.js and then small.js:

<script defer src="https://javascript.info/article/script-async-defer/long.js"></script>

<script defer src="https://javascript.info/article/script-async-defer/small.js"></script>

Browsers scan the page for scripts and download them in parallel, to improve performance. So in the example above both scripts download in parallel. The small.js probably finishes first.

…But the defer attribute, besides telling the browser “not to block”, ensures that the relative order is kept. So even though small.js loads first, it still waits and runs after long.js executes.

That may be important for cases when we need to load a JavaScript library and then a script that depends on it.

**The defer attribute is only for external scripts**

The defer attribute is ignored if the <script> tag has no src.

## [async](https://javascript.info/script-async-defer" \l "async)

The async attribute is somewhat like defer. It also makes the script non-blocking. But it has important differences in the behavior.

The async attribute means that a script is completely independent:

* The browser doesn’t block on async scripts (like defer).
* Other scripts don’t wait for async scripts, and async scripts don’t wait for them.
* DOMContentLoaded and async scripts don’t wait for each other:
  + DOMContentLoaded may happen both before an async script (if an async script finishes loading after the page is complete)
  + …or after an async script (if an async script is short or was in HTTP-cache)

In other words, async scripts load in the background and run when ready. The DOM and other scripts don’t wait for them, and they don’t wait for anything. A fully independent script that runs when loaded. As simple, as it can get, right?

Here’s an example similar to what we’ve seen with defer: two scripts long.js and small.js, but now with async instead of defer.

They don’t wait for each other. Whatever loads first (probably small.js) – runs first:

<p>...content before scripts...</p>

<script>

document.addEventListener('DOMContentLoaded', () => alert("DOM ready!"));

</script>

<script async src="https://javascript.info/article/script-async-defer/long.js"></script>

<script async src="https://javascript.info/article/script-async-defer/small.js"></script>

<p>...content after scripts...</p>

* The page content shows up immediately: async doesn’t block it.
* DOMContentLoaded may happen both before and after async, no guarantees here.
* A smaller script small.js goes second, but probably loads before long.js, so small.js runs first. Although, it might be that long.js loads first, if cached, then it runs first. In other words, async scripts run in the “load-first” order.

Async scripts are great when we integrate an independent third-party script into the page: counters, ads and so on, as they don’t depend on our scripts, and our scripts shouldn’t wait for them:

<!-- Google Analytics is usually added like this -->

<script async src="https://google-analytics.com/analytics.js"></script>

**The async attribute is only for external scripts**

Just like defer, the async attribute is ignored if the <script> tag has no src.

## [Dynamic scripts](https://javascript.info/script-async-defer" \l "dynamic-scripts)

There’s one more important way of adding a script to the page.

We can create a script and append it to the document dynamically using JavaScript:

let script = document.createElement('script');

script.src = "/article/script-async-defer/long.js";

document.body.append(script); // (\*)

The script starts loading as soon as it’s appended to the document (\*).

**Dynamic scripts behave as “async” by default.**

That is:

* They don’t wait for anything, nothing waits for them.
* The script that loads first – runs first (“load-first” order).

This can be changed if we explicitly set script.async=false. Then scripts will be executed in the document order, just like defer.

In this example, loadScript(src) function adds a script and also sets async to false.

So long.js always runs first (as it’s added first):

function loadScript(src) {

let script = document.createElement('script');

script.src = src;

script.async = false;

document.body.append(script);

}

// long.js runs first because of async=false

loadScript("/article/script-async-defer/long.js");

loadScript("/article/script-async-defer/small.js");

Without script.async=false, scripts would execute in default, load-first order (the small.js probably first).

Again, as with the defer, the order matters if we’d like to load a library and then another script that depends on it.

## [Summary](https://javascript.info/script-async-defer" \l "summary)

Both async and defer have one common thing: downloading of such scripts doesn’t block page rendering. So the user can read page content and get acquainted with the page immediately.

But there are also essential differences between them:

|  | **Order** | **DOMContentLoaded** |
| --- | --- | --- |
| async | Load-first order. Their document order doesn’t matter – which loads first runs first | Irrelevant. May load and execute while the document has not yet been fully downloaded. That happens if scripts are small or cached, and the document is long enough. |
| defer | Document order (as they go in the document). | Execute after the document is loaded and parsed (they wait if needed), right before DOMContentLoaded. |

In practice, defer is used for scripts that need the whole DOM and/or their relative execution order is important.

And async is used for independent scripts, like counters or ads. And their relative execution order does not matter.

**Service Worker:**

* A javascript that gets registered with the browser
* Stays registered with the browse even when offline
* Can load contents even without connection
* They cannot directly access the DOM
* Programmable network proxy : allows to control how the network request from the page is handled
* Terminated when not being used
* Make use of promises
* Require HTTPS unless on localhost

**Uses:**

* Caching assets and API calls
* Push Notification
* Background data sync/preload
* Used in progressive web apps (PWA)

**Service worker lifecycle and events**

1. Register
2. Install
3. Activate
4. Fetch

Reference URL: https://www.youtube.com/watch?v=ksXwaWHCW6k&ab\_channel=TraversyMedia